NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

LAYERED PATH PLANNING
FOR AN AUTONOMOUS MOBILE ROBOT

by

Timothy A. Haight

September 1994

Thesis Co-Advisors: Yutaka Kanayama
Craig W. Rasmussen

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
Layered Path Planning for an Autonomous Mobile Robot (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Haight, Timothy A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

In order to continue to improve the usefulness of robots, it is becoming increasingly important to have them act as autonomous agents. A significant step toward this object is autonomous motion planning. This research was conducted as part of a broader effort to empower Yamabico-11, a mobile robot under development at the Naval Postgraduate School, with ability to move autonomously. We believe that this problem is best attacked in layers.

This thesis is our proposal for the initial layer. Given a robot's current location and its goal location, we use the homotopy relation to reduce the infinite set of path choices into a more manageable and smaller set of path classes. Specifically, we solve the problem of how to enable a robot to autonomously identify and label these classes of paths.

We begin by decomposing the robot's operating environment into a collection of convex pieces called cells. The cells are transformed into a graph by adjacency. We show that every simple path on the graph corresponds to a unique simple homotopy class in the robot's world. We then search the graph to give each class a symbolic representation which also provides intermediate path planning clues. Subsequent layers can use these clues to form a more detailed plan.

We implement the cell decomposition, graph transformation, and path class labelin · as C programs, and preprocess them on a Unix workstation. The resulting data structures are then comp· ·d and linked into the robot's kernel. All implementation has been integrated into the model-based mo··le robot language (mml) used by Yamabico-11.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous vehicle, mobile robot, motion planning	122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

LAYERED PATH PLANNING FOR AN AUTONOMOUS MOBILE ROBOT

Timothy A. Haight
Captain, United States Army
B.S. , United States Military Academy

Submitted in partial fulfillment of the requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE
MASTER OF SCIENCE IN MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Author: _____
Timothy A. Haight

Approved By: _____
Yutaka Kanayama, Thesis Co-Advisor

_____
Craig W. Rasmussen, Thesis Co-Advisor

_____
Ted Lewis, Chairman, Department of Computer Science

_____
Richard Franke, Chairman, Department of Mathematics

ABSTRACT

In order to continue to improve the usefulness of robots, it is becoming increasingly important to have them act as autonomous agents. A significant step toward this object is autonomous motion planning. This research was conducted as part of a broader effort to empower Yamabico-11, a mobile robot under development at the Naval Postgraduate School, with ability to move autonomously. We believe that this problem is best attacked in layers.

This thesis is our proposal for the initial layer. Given a robot's current location and its goal location, we use the homotopy relation to reduce the infinite set of path choices into a more manageable and smaller set of path classes. Specifically, we solve the problem of how to enable a robot to autonomously identify and label these classes of paths.

We begin by decomposing the robot's operating environment into a collection of convex pieces called cells. The cells are transformed into a graph by adjacency. We show that every simple path on the graph corresponds to a unique simple homotopy class in the robot's world. We then search the graph to give each class a symbolic representation which also provides intermediate path planning clues. Subsequent layers can use these clues to form a more detailed plan.

We implement the cell decomposition, graph transformation, and path class labeling as C programs, and preprocess them on a Unix workstation. The resulting data structures are then compiled and linked into the robot's kernel. All implementation has been integrated into the model-based mobile robot language (mml) used by Yamabico-11.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

In order to improve the usefulness of robots, it becomes increasingly important to have them act as autonomous agents. A significant subproblem of this objective is motion planning. Autonomous motion planning is a broad area including a diverse set of topics. Among these is the problem of enabling a mobile vehicle to relocate itself from one configuration to another while avoiding obstacles along its path.

This research was conducted as part of a broader effort to empower Yamabico-11, a mobile robot under development by students and faculty of the Naval Postgraduate School, with the ability to move autonomously. We propose to simplify this difficult problem by attacking it in layers. The highest and most abstract layer partitions the set of paths into equivalence classes. A specific class is then selected and used by lower layers where the detailed motion plan is formed. The lowest and final layer interprets the detailed motion plan into specific control instructions for the mobile robot, where movement is realized.

A. PROBLEM STATEMENT

Given an initial and goal configuration for a mobile robot, we want to partition the infinite set of path choices into a more manageable and general set of path classes. Each class must be uniquely and unambiguously identified in a manner consistent with its topology. Additionally, the class names must provide useful motion planning information for lower levels. Finally, the model used to represent the robot's world and the set of path classes must be integrated into the global motion planner and the model-based mobile robot language used by Yamabico.

B. ASSUMPTIONS

The first assumption is that the robot will be working in a familiar environment with complete knowledge of the topology. It will not, however, use external references to guide its motion, such as following marked or predescribed paths. We also assume that the robot

has perfect knowledge of its configuration (location and orientation). This is accomplished through odometry control, and if necessary, self-correction using internal sensors.

All work will be done in two dimensions. We do not assume that the robot or its operating environment are restricted to two dimensions, but we do assume that the problem can be solved by considering the projection of the robot and its environment onto the XY-plane.

The last assumption is that some of the work can be preprocessed. The preprocessing includes representing the operating environment, representing the path classes, and building the model used by the upper layers of the motion planning process. This preprocessing frees the robot to perform the real-time calculations necessary to realize motion and operate its internal sensors.

C. APPROACH

We begin by decomposing the world into a collection of path-connected convex cells whose union is exactly the robot's free space. Information about the connectivity of the cells is transformed into a graph which is searched to find path classes. The global motion planner selects a class, and then uses the graph to determine the sequence of cells through which the robot must move. This approach simplifies the task of motion planning in two areas. First, by attacking the problem in layers, we decrease the number of choices that must be considered by the global motion planner. Second, since the robot will move from convex cell to convex cell, the computation required for intracell motion planning should be reduced. This frees the processor for lower level tasks.

D. YAMABICO-11

Although the theory and essence of this work applies to autonomous motion planning for any mobile robot, it will first be implemented on Yamabico-11. Yamabico-11 has a single axis with two fixed wheels which are independently driven by separate motors. Additionally, it has four shock absorbing, free-moving, caster wheels for stability. Control of the robot is accomplished by a single SPARC processor with 16 Mbytes of main

memory. Twelve 40KHz ultrasonic senors are used by the robot to verify odometry and avoid unexpected obstacles.

Yamabico is currently located in the Computer Science Robotics Laboratory on the fifth floor of Spanagel Hall at the Naval Postgraduate School. Spanagel Hall is a typical academic building. The fifth floor consists of a long hallway with classrooms and offices to either side, and a large computer lab at the east end. Most of the testing is conducted in the hallway and foyer immediately outside of the lab. The hallway has no indigenous obstacles, so wooden boxes are temporarily placed wherever an obstacle is desired.

All implementation programs are written in ANSI C and compiled using the GNU Project C Compiler. User access to implementation programs is provided through the model-based mobile robot language (mml), a high level language developed by students at the Naval Postgraduate School. Specific details of the hardware and software systems of Yamabico can be found in [Yama93] and [Yama94].

E. THESIS ORGANIZATION

The next chapter provides formal definitions used throughout the thesis. It introduces the reader to some of the basic elements of topology, and describes the primary tool we will use in the initial path planning layer. Chapter III is a discussion of two properties of a robot's operating environment that have served as the basis for previous motion planning methods. The chapter includes an introduction to the common data structures and recent research of both properties. Although these properties are not strictly related, we combine their discussion in this chapter to separate them from our proposal. Chapter IV is a detailed presentation of our proposed method. Here we present the underlying idea and provide examples of how we use the connectivity of the robot's world to reduce the path planning problem. We also talk about a potential weakness in our plan and offer two solutions. In Chapter V we describe the implementation of this method on Yamabico-11, to include the geometric model, data structures and algorithms. We end the

thesis by analyzing the method we have chosen, and by mentioning additional and supplementary topics of research.

II. TOPOLOGY

Before we discuss the issue of motion planning, we need to give some precise meaning to the concepts that provide the basis for our proposal. In accordance with the previously stated assumptions, we will restrict the discussion in this chapter to the Euclidean plane. Additionally, we consider a robot to be a point unless explicitly stated otherwise. This chapter is not a complete lesson in topology, but rather a formal introduction of those definitions needed later. The reader familiar with this area can skip the chapter without loss of continuity.

A. TOPOLOGICAL SPACE

1. Definitions

From [Cr78] we take the standard definition of a *topology* for a set X as a family T of subsets of X satisfying the following properties:

1. The set X and the empty set $\varnothing$ are in T.
2. The union of any family of members of T is in T
3. The intersection of any finite family of members of T is in T

A *topological space* is a pair (X,T) where X is a set and T is a topology for X. If there is no ambiguity, the topological space can be referred to simply as X. A space X is *connected* if it is not the union of two disjoint, nonempty open sets. Intuitively, this means that X can best be viewed as "one piece", and is in some sense indecomposable. A related idea, and one which is more suitable to our purposes is that of *path connectedness* [GaGr83].

Let X be a topological space, and let x_0 and $x_1 \in X$. A *path* Π in X from x_0 to x_1 is a continuous function $f:[0,1] \rightarrow X$ such that $f(0) = x_0$ and $f(1) = x_1$. We say that X is *path connected* if for every pair of points x_0 and x_1 in X, there exists a path between them. Additionally, If a space is path connected, then it is also connected [GaGr83].

Two characterizations of sets which are needed for later definitions are whether a set is open or closed, and whether a set is bounded or unbounded. A set is *closed* if and only if it contains its boundary. Additionally, the complement of a closed set is *open* which implies that a set is open if and only if it contains none of its boundary. Since a set may contain only a portion of its boundary, it may be neither open nor closed. We give the definition of a bounded set by using the intuitive notion of distance. A set is *bounded* if the distance between any two of its members is finite. A set that is not bounded is said to be *unbounded*. [Ki89]

Finally, we introduce the concept of a hole. The Jordan Curve Theorem states that a simple closed curve C in the Euclidean plane separates the plane into two open connected sets with C as their common boundary. Exactly one of these sets is bounded.[Cr78] We define a hole to be one of the open connected sets. We say that the hole is *normal* if it is bounded, and *inverted* if it is unbounded. Sometimes it may be useful to consider the hole along with its boundary, but generally we refer to them separately.

2. The Robot's Space

For this research, we consider the robot's space to be the Euclidean plane with holes. We allow an unlimited, but finite, number of normal holes, which are obstacles for the robot. We also allow one inverted hole, which if present, defines the robot's outer limits. We assume that the boundary of all holes are simple polygons. Furthermore, we consider the boundary of a hole to be directed curve which when traversed, puts the hole to the left. This directed boundary naturally defines the neighbors of a vertex to be the *next vertex*, and the *previous vertex*. The *free space*, or the robot's operating space, is the complement of the union of all the holes. We call the free space, together with the set of holes, the robot's *world*.

We also consider paths to be directed curves with natural direction from $f(0)$ to $f(1)$. We say that $f(0)$ and $f(1)$ are the endpoints, and that the path joins them. We refer to $f(0)$ as the start or initial position, and $f(1)$ as the goal. Figure 1, on page 7, is an example of a world

with two normal holes h_1 and h_2; one inverted hole h_3; and three paths π_1, π_2, and π_3 from S to G.

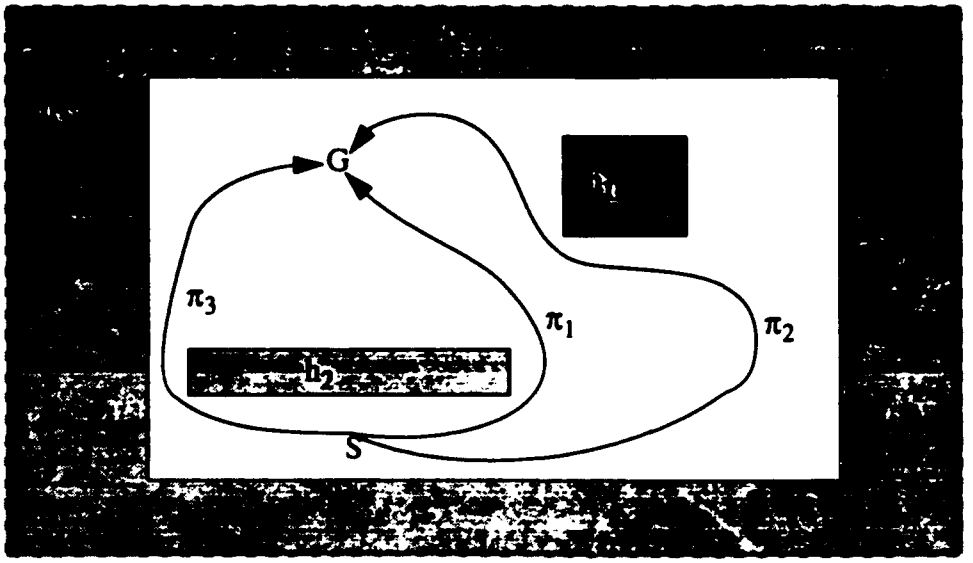

Figure 1: Topological Space with 3 holes and 3 paths

B. HOMOTOPY

It should be clear that, in any connected space, the set of paths between any two points is infinite. In order to simplify the problem of choosing a path we want to group paths that are, in some sense, alike. Before we give a formal definition, we present an intuitive idea of what makes two paths similar. In Figure 1, we see that paths π_1 and π_2 are somewhat similar in that they both go to the right of h_2 and to the left of h_1. Another observation is that there is no hole "between" them. Notice, however, that h_2 is between paths π_1 and π_3. Based on these observations we might conclude that π_1 and π_2 should be grouped together, and π_3 should be in a group by itself. The relation of *homotopy* provides a formal method for making these groupings.

Consider two paths in the robot's world, f:[0,1] and g:[0,1], with common endpoints. We say that f is homotopic to g, written $f \cong g$, provided there is a continuous function H:[0,1]×[0,1]→X which satisfies these equations:

$$H(t,0) = f(t) \ \forall \ t \in [0,1] \qquad \text{(Eq 2.1)}$$

$$H(t,1) = g(t) \ \forall \ t \in [0,1] \qquad \text{(Eq 2.2)}$$

$$H(0,x) = f(0) = g(0) \qquad \text{(Eq 2.3)}$$

$$H(1,x) = f(1) = g(1) \qquad \text{(Eq 2.4)}$$

In other words, H is a function that allows us to continuously deform one path into the other without crossing an obstacle. Furthermore, homotopy defines an equivalence relation on the set of paths which partitions them into a collection of homotopy classes. [Cr78] We will use this relation to reduce the problem of path selection by considering a finite set of homotopy classes rather than an infinite set of paths.

III. PROXIMITY AND VISIBILITY

Two fundamental properties of the robot's operating environment are proximity and visibility. Each has been used as a basis for previous path planning methods. In this chapter we will introduce and discuss the common data structures used to capture these properties, and we will briefly present some of the algorithms used to build and employ them.

A. PROXIMITY

Proximity tells us the obstacle, or portion of an obstacle, to which the robot is closest. This is important locally when trying to avoid obstacles since the closest one presents the most immediate danger. However, proximity information is more commonly used globally when planning for obstacle-avoiding paths. In this section we will focus on the global aspects of proximity and how they can be used for autonomous path planning.

1. Voronoi Diagrams

The *Voronoi diagram* was first used in 1975 by Shamos and Hoey [ShHo75] as a means of representing proximity for a finite set of points in the Euclidean Plane. It has since become the elementary data structure for representing proximity, and continues to have broad application in the field of computational geometry. Generally, the Voronoi diagram $VD(O)$ of a set O of n objects in a space W is a subdivision of this space into maximal regions so that all points within a given region have the same nearest neighbor in O with regard to a general distance measure d [PrSh85]. The Voronoi Diagram is divided into two parts: *Voronoi boundaries* $B(o_i,o_j)$ between two objects o_i and o_j, and *Voronoi regions* $V(o_j)$ of an object o_j. They are described by the following equations:

$$B(o_i,o_j) = \{p \in W \| d(p, o_i) = d(p, o_j)\} \qquad \text{(Eq 3.1)}$$

$$V(o_j) = \{p \in W \| \forall j, j \neq i;\ d(p, o_i) < d(p, o_j)\} \qquad \text{(Eq 3.2)}$$

The first Voronoi diagram, often called the *Euclidean Voronoi diagram (EVD)*, was defined over a set of points $O=\{p_1,...,p_n\}$, using the Euclidean distance function in $\mathcal{R}^2$. In this case, the Voronoi Boundary $B(p_i,p_j)$ is the perpendicular bisector of p_i and p_j. When only two sites are considered, $B(p_i,p_j)$ separates the plane into three regions: all points which are closer to p_i than to p_j, all points which are closer to p_j than to p_i, and the boundary itself. If we denote the half-plane which contains the set of all points closer to p_i than to p_j as $H(p_i,p_j)$, then the Voronoi Region $V(p_i)$ is the polygon formed by the intersection of the n-1 half-planes $H(p_i,p_k)$, for all $p_k \in O \setminus p_i$. The EVD is the union of $V(p_i)$ for all $p_i \in O$.

Let us now consider the construction of the EVD. From its definition we immediately see that one approach would be to build each polygon individually. In this case, we can form the intersection of the n-1 half-planes in time $O(N^2)$, which suggests that the EVD can be constructed in time $O(N^3)$. However, the construction of the polygons can be improved to O(NlogN) by using a simple divide-and-conquer approach, lowering the bound of constructing the EVD to $O(N^2 \log N)$. In fact, the entire EVD can be attacked by a divide-and-conquer strategy which yields an optimal Θ(NlogN) algorithm presented by Preparata and Shamos [PrSh85]. The idea behind this algorithm is to divide the set of obstacles into roughly equal parts by median x-coordinate, construct the Voronoi Diagram for both parts recursively, and then merge the two pieces to form the complete Voronoi Diagram. The bound relies on the fact that the division and merge steps each take O(N). We refer the reader to [PrSh85] for the details of the algorithm and a proof of its correctness.

We now briefly mention two important properties of the EVD, but again refer the reader to [PrSh85] or any other text on computational geometry for a complete catalog and discussion. First, an embedding of the EVD yields a planar straight-line graph. This allows for O(N) storage of the complete proximity information. Second, the straight-line dual of the EVD is a triangulation of O. This dual, called the *Delaunay triangulation*, is obtained

by adding a line between two points if their Voronoi regions share an edge. This fact not only implies that the Voronoi diagram can be used to solve other problems in the field, but it is also used in some alternative construction algorithms.

By examining the definition of the Voronoi diagram, we see that the EVD can be generalized in three areas: the space W, the objects O, and the distance measure d. A fourth, but less common, generalization is to consider the set of points that are closer to any k-subset of O.Voronoi diagrams used for motion planning are often generalized with respect to O, since the obstacles in a robot's world are rarely points. However, Voronoi diagrams defined for higher dimension or abstract distance functions have also been used. An interesting generalization under our assumptions is constructed in $\mathcal{R}^2$ using Euclidean distance, with O being the set of edges and vertices of the polygonal obstacles. In this case, the shape of the Voronoi boundary between two elements of O depends on the type of objects considered. Let e_i and e_j be edges, and let v_l and v_m be vertices of O. As in the EVD, $B(v_l,v_m)$ is the perpendicular bisector of v_l and v_m, and $B(e_i,e_j)$ is the bisector(s) of e_i and e_j. However, $B(e_i,v_l)$ or $B(v_l,e_i)$ is the parabola defined by the focus v_l and the directrix e_i. The Voronoi region $V(e_i)$ or $V(v_l)$ remains the intersection of the n-1 half-planes defined by the Voronoi boundaries of e_i or v_l. More importantly, the Voronoi region of an obstacle is the union of the Voronoi regions of its edges and vertices. The construction of this generalized Voronoi diagram is not as straight-forward as the EVD, but is still optimally computed in time Θ(NlogN). In the next section we present an alternative construction algorithm for the generalized Voronoi diagram, and another generalization of the EVD with application to robot motion planning.

2. Recent work

Although the divide and conquer algorithm is well-suited for constructing the EVD and some basic generalized Voronoi diagrams, it may not be practical in all cases. Here we examine a technique for constructing the Voronoi diagram by a randomized incremental addition of the obstacles. First, we take the generalized definition of a Voronoi Diagram

from [Kl89]. For a set of obstacles $O=\{o_1,...,o_n\}$, define $J(o_i,o_j) = J(o_j,o_i)$ to be a bisecting curve separating sites o_i and o_j. Let $D(o_i,o_j)$ and $D(o_j,o_i)$ be the two domains separated by $J(o_i,o_j)$ such that any point in $D(o_i,o_j)$ is closer to o_i than to o_j. Additionally, we must choose exactly one of $D(o_i,o_j)$ or $D(o_j,o_i)$ to include $J(o_i,o_j)$. In the case of the EVD, $J(o_i,o_j) = B(p_i,p_j)$ and $D(o_i,o_j) = H(p_i,p_j)$. The Voronoi region of o_i with respect to O, $VR(o_i,O)$, is the intersection of the $D(o_i,o_k)$, for all $o_k \in O \setminus o_i$, and the Voronoi diagram of O, $V(O)$, is the union of the boundaries of the $VR(o_i,O)$, for all $o_i \in O$. This definition appears very similar to that of the EVD, but, note that we have made no reference to the definition of distance or the type of obstacles in O.

Mehlhorn, Meiser, and O'Dúnlaing [MMO91], propose to build $V(O)$ incrementally. Let $R \subseteq O$ be the set of sites already added to $V(O)$. They maintain two data structures: The Voronoi diagram $V(R)$ stored as a planar graph, and the conflict graph $G(R)$. The conflict graph consists of vertices which are the edges of $V(R)$ and the obstacles in O-R. There is an edge (conflict) in $G(R)$ between vertices corresponding to an edge e of $V(R)$ and an obstacle $o_i \in O$-R if and only if e has a nonempty intersection with $VR(o_i,R \cup o_i)$. Now for each obstacle o_i added to R, they show that only those edges in conflict with o_i need to be changed when updating the $V(R)$. The complexity of updating $V(R)$ and $G(R)$ combine to produce an algorithm which runs in time O(NlogN).

To complete this section we want to mention an interesting generalization of the EVD with application to robot motion planning. Chew and Kedem's [ChKe90] approach to safe motion planning is based on the intuitive idea that motion along the Voronoi boundaries provides maximal clearance. They construct a Voronoi diagram considering polygonal obstacles in $\mathcal{R}^2$, but use a convex distance function defined by the geometrical shape of the polygonal robot. Since the shape of the robot is not a Euclidean circle, the distance between two points changes as the robot rotates. They track the effect of these changes by plotting the Voronoi diagrams in (x,y,θ) space. They show that building the

initial diagram takes O(KNlogKN), where K is the number of sides of the robot. Updating the Voronoi diagrams in three-space, however, raises the complexity to $O(K^4NlogN)$.

B. VISIBILITY

Consider the robot's world W, and the associated free space F. We say that two points, x and y, are *visible* if they can be connected by a line segment xy such that $xy \subseteq F$. In other words, xy cannot intersect any of the holes in W except possibly at a boundary. If x and y are visible, we also say that one *sees* the other.Visibility is important to motion planing for two reasons. First, if the robot can see its goal then, intuitively, the problem of motion planning may be simplified. Second, the shortest path between a robot and its goal can be found by examining the visibility relationship between the robot, the goal, and the vertices of the polygonal holes. In this section we will examine a common data structure used to represent visibility and its application to motion planning.

1. Visibility Graphs

The visibility graph of a polygon is a graph on its vertices such that two vertices are joined by an edge if and only if they are visible. An upper bound for the number of edges in the visibility graph occurs when the polygon is convex. In this case, the graph will have $\binom{n}{2}$ edges since every vertex in the polygon can see every other vertex. An immediate lower bound for n is obtained by observing that the edges of the original polygon are also edges in the visibility graph. O'Rourke [OR87], however, shows that a true lower bound is $2n$-3, as any polygon will have at least 3 convex vertices.

As a motion planning tool, we can extend the concept of the visibility graph to the case of a polygon with holes. As before, the vertices of the graph are those of the polygons, and the graph edges represent the pairs of vertices that are mutually visible. In this case, however, we should include the initial and goal position of the robot as vertices in the visibility graph, and connect them to other vertices as appropriate. We can conveniently use this structure to determine the shortest obstacle-free path from the robot's current position

to its goal. For simplicity, we will assume that the robot is a point, and can therefore pass between obstacles along a visible path. We first notice that if the start and goal positions are joined by an edge in the visibility graph, then the shortest path is simply the straight line joining the two. If they are not joined by an edge, Alt and Welzl [AlWe89] show by a simple geometric argument that the shortest path is a polygonal chain whose vertices are vertices of the obstacles. Since the robot's path must be obstacle-free, the shortest path polygonal chain must also avoid obstacles. Consequently, pieces of the polygonal chain correspond to edges of the visibility graph. An assignment of Euclidean distance to the graph arcs allows the shortest path to be computed using Dijkstra's algorithm in time $O(N^2)$.

We now examine the complexity of building the visibility graph. Let V be the set of obstacle vertices, and let E be the set of obstacle edges. A straightforward approach would be to examine each pair of vertices in V to see if the line segment joining them intersects an edge in E. If $N=|V|=|E|$, then the algorithm compares $O(N^2)$ vertex pairs with $O(N)$ edges, and will run in time $O(N^3)$. We observe that not all pairs in V need to be considered, since vertices from the same obstacle will not be incident in the visibility graph unless they are connected by an edge in E. Still, this algorithm is $O(N^3)$ in the worst case. We will discuss in the next section how to first improve the algorithm to run in time $O(N^2 \log N)$, and then how to improve it to run in $O(N^2)$. We will also mention an interesting alternative to the visibility graph which can be used to solve the shortest path problem.

2. Recent work

The definition of the visibility graph and its optimal construction have remained fairly stable since the presentation by O'Rourke [OR87]. At that time, however, little was known about their characterizations from a graph-theoretical viewpoint. Most of the recent work concerning visibility graphs involves understanding these characterizations. We refer the reader to [LMW87] or [An92] for these results.

Still, it is worth noting the method used to improve the algorithm for constructing the visibility graph. The original idea is attributed to [Lee78], but is presented in [AlWe89] along with an improvement. We will present the main ideas of both algorithms here. Let p,q ∈ V, the set of all obstacle vertices. Define $vis_d(p)$ to be the open ray emanating from p in direction d, and assign it a value based on which obstacle edge it encounters first. Now, pick an arbitrary direction d_0 and initialize $vis_d(p)$ for all $p \in V$. Next rotate d until $d=\pi+d_0$, while updating $vis_d(p)$. This continuous rotation is discretized by noting that $vis_d(p)$ is only updated if d is the same orientation as a line determined by p and some other vertex q. Additionally, the value of $vis_d(p)$ as it changes determines whether the edge pg belongs in the visibility graph. By sorting all pairs of vertices in V by the slope of their connecting lines, $vis_d(p)$ is only updated $\binom{n}{2}$ times. Determining the initial $vis_d(p)$ for all $p \in V$ is accomplished easily in time $O(N^2)$, but this can be improved to O(NlogN). Sorting the vertex pairs takes $O(N^2 \log N)$ time. Each update of $vis_d(p)$ for the $O(N^2)$ pairs can be processed in constant time. Therefore, the total running time of the algorithm is $O(N^2 \log N)$. The final improvement comes from the idea that a complete sorting of the vertex pairs is not necessary; a topological ordering is sufficient. Since this can be accomplished with amortized complexity of $O(N^2)$, the entire algorithm is reduced to $\Theta(N^2)$. This is optimal as the visibility graph may have $O(N^2)$ edges.

We close this section by mentioning a variation of the shortest path problem that does not use the visibility graph. For a set of obstacles O and a point s, the *shortest path map*, denoted SPM(O,s), is a partition of the robot's free space into regions such that any two points p and q lie in the same region if and only if the shortest paths from s to p and from s to q touch the same sequence of vertices. If the initial position of the robot is s, and its goal is t, we can solve the shortest path problem as follows: Construct SPM(O,s), and then locate the region containing t. For the case of polygonal obstacles it has been shown

that this can be done in O(N(K+logN)), where K is the number of obstacles. For large K this is $O(N^2)$, otherwise it is O(NlogN).

IV. CONNECTIVITY

Kanayama [Ka94] discusses the sensitivity of visibility, proximity, and connectivity to small continuous changes of the robot's operating environment. It is his conclusion that visibility is globally affected, proximity is locally affected, and connectivity is unaffected. Because of this, we prefer path planning methods which rely more on connectivity, and less on proximity and visibility. In the first part of this chapter we present a means of labeling homotopy classes considering only connectivity, and using minimal information. We then present a method of transforming the abstract problem of identifying the classes into a straight-forward graph search problem by adding additional information to the class names.

A. IDENTIFYING HOMOTOPY CLASSES

Consider a robot's world W with a finite number n of normal holes. If there is no inverted hole, we assume the presence of one of sufficient size to minimize its influence on the proximity information of W. We also assume that the robot's free space is path connected. Define a fence L to be a loop free curve connecting two holes which does not intersect a hole or another fence except at its endpoints. The idea is to add a maximal number of fences to the world, while maintaining the connectedness of the free space, so that we can use them to identify homotopy classes. If we consider the holes as nodes of a planar graph, and the fences as arcs, we know by Euler's formula that we can add n fences to the n+1 holes without dividing the freespace. We call this collection of holes and fences a *connected world*. Clearly, if the world has $n \geq 2$ holes, then the construction of the connected world is not unique. Figure 2, on page 18, is an example of a connected world with two normal holes, and one inverted hole. Here, we have added the two fences a and b indicated by the dotted lines. Figure 2 also shows three paths from S to G, indicated by the solid lines. Each path represents a different path classes. Notice that two of the path classes are labeled by naming the fence that they cross, and the third by ε since it does not cross a fence.

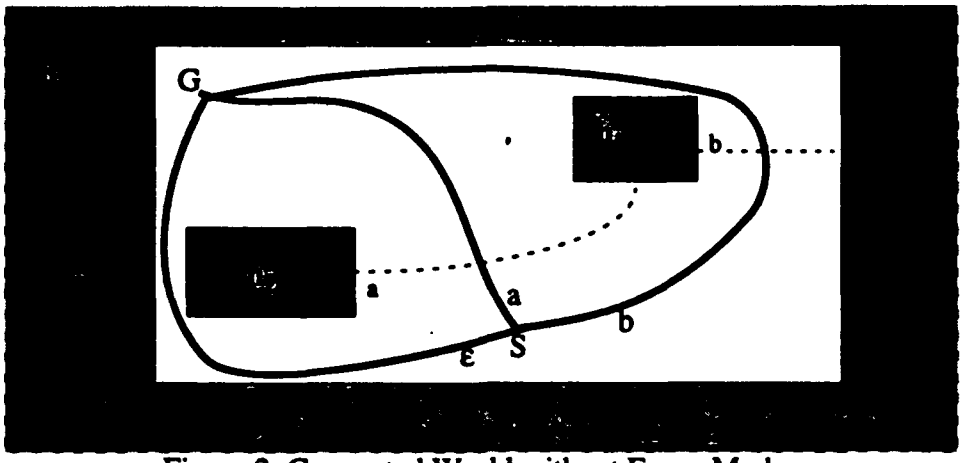

Figure 2: Connected World without Fence Modes

If, however, we add a fourth path shown by the dashed line in Figure 3, then labeling the paths by fence crossing alone is not sufficient. Our solution is to redefine a fence so that it has two sides; a *plus side* and a *minus side*. If a path intersects a fence from the plus side, we say that it has *plus intersecting mode*. Likewise, if a path intersects a fence from the minus side, we say that it has *minus intersecting mode*. Now, we relabel each path class by the fence and intersecting mode. We call this class name the *fence crossing sequence*. Kanayama [Ka94] proves that two paths will have the same fence crossing sequence if and only if they are homotopic.

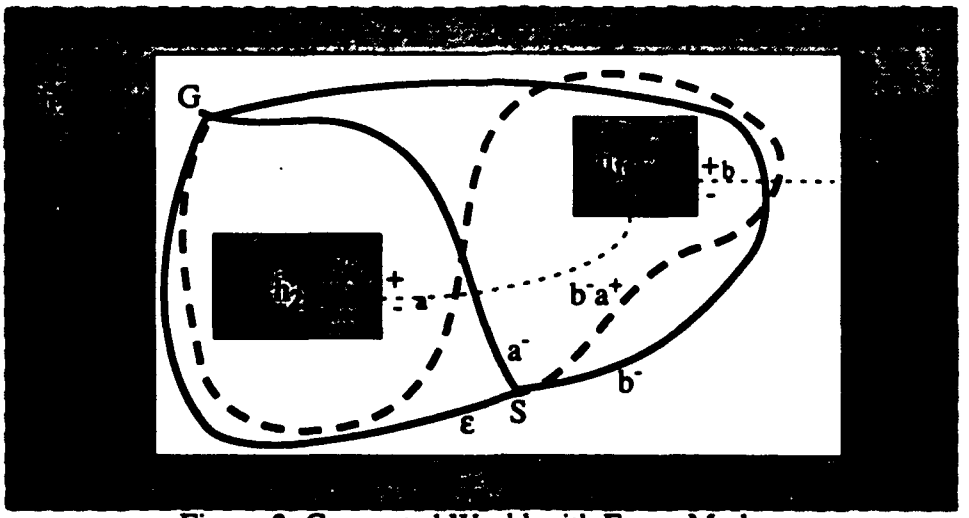

Figure 3: Connected World with Fence Modes

Although this labeling method relies solely on the connectivity of the world, it provides very little path planning information. Our goal is a naming method which gives intermediate motion clues in addition to unique path class labels. In this case, there are just too many options for the robot to consider between fence crossings. Also, we want the robot to be able to generate the class names automatically. Unfortunately, there are no obvious rules which allow valid fence crossing sequences to be considered, and invalid sequences to be rejected. In the rest of this chapter, we describe how to augment the fence crossing sequence to meet these goals.

B. FREE SPACE DECOMPOSITION

Latombe [La91] proposes an approach to motion planning which he calls *exact cell decomposition*. This method divides the free space of the operating environment into a collection of non-intersecting regions, called cells. They are constructed so that intracell motion planning is an easier problem than motion planning within the entire free space. Since motion planning within a convex polygon eliminates the issue of visibility, and minimizes the issue of proximity, convex cells are desired. A decomposition of the free space in which all cells are convex, referred to as a *convex polygonal decomposition*, is preferred. While the basis for the following method is certainly not new [Chz87], we believe that its application extends beyond any presented in current literature.

Let W be the subset of $\mathcal{R}^2$ which is the robot's world, or the space that we need to decompose. Let H be the set of holes within W, and let V be the set of all vertices of H. An efficient means of achieving a convex polygonal decomposition of W is to divide the free space by a series of parallel lines placed at certain critical points along the holes of H. This is accomplished by sweeping a line L_α of constant orientation α across W. At each convex vertex of V, we extend the line in both directions (α and α+180) until it intersects a hole in W. The extensions can be characterized by the geometry of the vertex.

We say that vertex v_i is *less than* vertex v_j, written $v_i < v_j$, if L_α intersects v_i before v_j. In this case, we can also say that v_j is *greater than* v_i ($v_j > v_i$). We say that vertex v_i *equals*

v_j, written $v_j = v_i$, if L_α encounters both vertices simultaneously. Recall from a previous discussion the definition of a vertex's next and previous vertices. Define a *minimal (maximal) extreme vertex of h* with respect to L_α to be a vertex which is less than (greater than) both its next and previous vertices. Define an *interior vertex* of h with respect to L_α to be any vertex that is not extreme. An interior vertex is called *up* if it is above the obstacle to which it belongs, and *down* if it is below. Furthermore, classify an extension of L_α as *up* if it is extended in the α direction, and *down* if it is extended in the α+180 direction.

Now consider v, a vertex of a hole h. If v is an extreme vertex of h, then L_α will be extended both upward and downward. If v is an interior vertex of h, then L_α will be extended upward or downward, but not both. L_α is extended upward if v is up interior, and downward if v is down interior. The sole upward or downward extension from interior vertices is a consequence of L_α immediately intersecting h in the other direction.

Figure 4, on page 21, illustrates the effect of sweeping L_α across a world with one normal hole and one inverted hole. In this example, the leading edge defined by vertices v_1 and v_6 is parallel to L_α; therefore, v_1 and v_6 are interior vertices even though they are at the extreme of h_1. When we extend L_α from v_1, it intersects h_1 immediately in the downward direction, and h_2 in the upward direction. Conversely, the extensions from v_6 intersect h_1 immediately in the upward direction, and h_2 in the downward direction. Vertices v_3 and v_5, however, are exterior vertices, and have extensions in both directions. This example also illustrates why we do not extend L_α from concave vertices. Note that vertices v_2 and v_4 are concave with respect to h_1, yet they are convex with respect to the free space of W. We can save work by skipping these vertices with the realization that they already contribute to the goal of obtaining convex polygonal cells. The result of sweeping L_α across W is a collection of convex polygonal cells, which we will simply call *cells*, and a collection of extensions, which we will call *fences*.

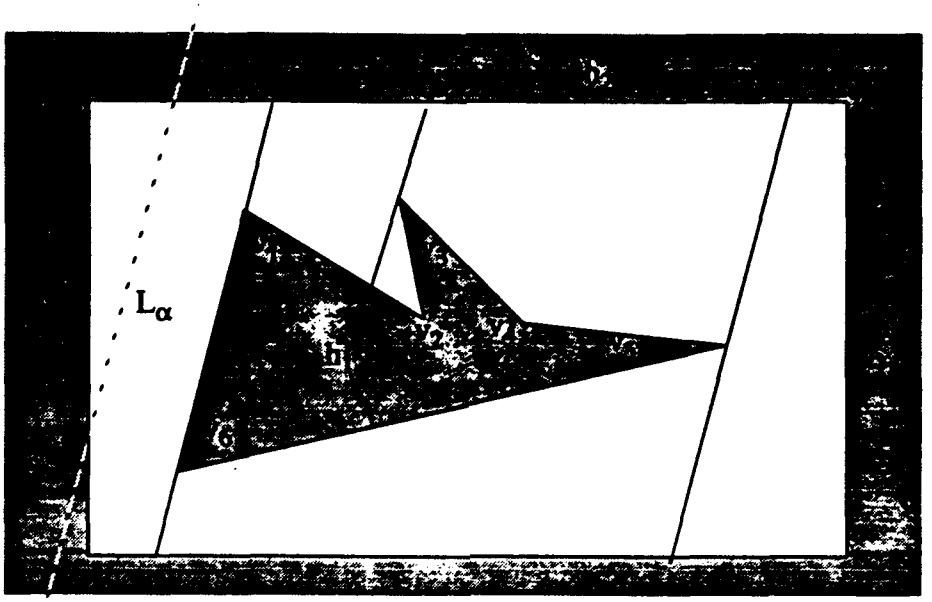

Figure 4: Results of sweeping L_α across W

A natural question to ask is whether the choice of orientation for L_α is significant, and how changing the orientation affects the cell decomposition. Clearly, the number and shape of the cells may differ significantly for various choices of α. Consider the simple world in Figure 5, on page 22, which shows two separate decompositions; one produced by sweeping L_0 (thin dotted line) and the other by L_{90} (thick dashed line). The L_0, or horizontal, decomposition contains five cells, while the L_{90}, or vertical, decomposition contains seven. This is intuitively explained by realizing that sweeps of different orientations capture different spatial information about the world. The top and bottom cells of the horizontal decomposition contain no vertical information about the holes h_1 and h_2, whereas the middle cell of the vertical decomposition contains no horizontal information. By this we mean that knowing an object is located in the top (middle) cell of the horizontal (vertical) decomposition is not sufficient to describe its location with respect to the left or right (top or bottom) of holes h_1 and h_2.

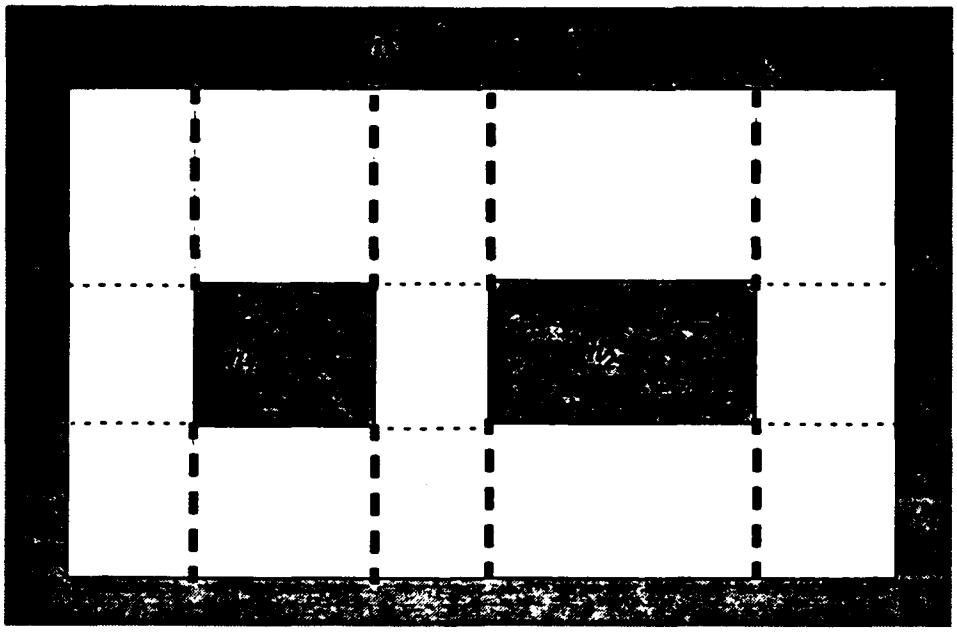

Figure 5: Effect of Changing Orientation of L_α

In general, any choice of L_α might create *ambiguous cells*, or those that lack geometric information with respect to L_α's orthogonal axis. The solution is to be able to identify these cells and then supplement them with additional information if necessary. There are two situations which cause the generation of ambiguous cells. We will describe both here, but defer the discussion of how they are treated until after we introduce the appropriate data structure and its application.

The first form of ambiguous cell is that created between the trailing extreme vertex (or parallel edge) of one hole, and the leading extreme vertex (or parallel edge) of another. In a world with multiple holes, we can expect this situation to occur often. Two special cases of this form are always present when L_α intersects the first normal hole, and when it leaves the last normal hole. In a world with a convex inverted hole, these appear as the first and last cell created. Fortunately, they are easy to detect and easy to fix. Figure 5, on page 22 contains multiple instances of this problem. The top and bottom cell of the horizontal decomposition and the left-most and right-most cell of the vertical decomposition are special cases. The middle cell of the vertical decomposition is a standard example.

The second form of ambiguous cells occurs when L_α encounters leading or trailing extreme vertices (or parallel edges) from multiple holes simultaneously. The top and bottom cells of the horizontal decomposition above are examples. In this case, however, they are also ambiguous cells of the first form. We are mainly concerned, though, with those cells that are only of the second form. These are less common, but still easy to identify, and in most cases can be eliminated by carefully choosing L_α.

It is not immediately apparent, then, whether one choice for L_α is better than all others. Clearly, we would like to reduce the number of ambiguous cells by making a good choice for L_α, but we do not want to spend too much effort finding it. We believe that this question requires further investigation. In the interim, we will always perform a vertical sweep. Sweeps of other orientations can be achieved by rotating the world, performing a vertical sweep, and then rotating the world back to its original position.

C. CONNECTIVITY GRAPH

The parallel cell decomposition induces a graph which we can use to extract some motion planning information. Latombe [La91] defines the *connectivity graph* for a convex polygonal decomposition by associating each cell with a node and connecting two nodes with an edge if and only if the corresponding cells are adjacent. Two cells are adjacent if they share a common fence. His use of the graph is restricted to determining the existence of an obstacle-free path within the robot's world. We propose to expand its use and to apply it as a tool for generating all simple homotopy classes.

Before we can use the connectivity graph, we must first assign labels to the nodes. Although any arbitrary assignment of labels is sufficient, we will use a more orderly approach. We will label each cell in the order of its inception. So for a vertical sweep, cells are labeled from left to right, and if two or more cells are created simultaneously, they are labeled from top to bottom. Additionally, there are two cells of special interest in every world: the one where the robot is currently located, and the one which contains its goal. They will be denoted by c_{init} and c_{goal}, respectively. In some cases, they may be the same.

Figure 6 is an example of the vertical parallel cell decomposition of a world with four normal holes and one inverted hole. Its associated connectivity graph is shown by the overlayed dotted line.

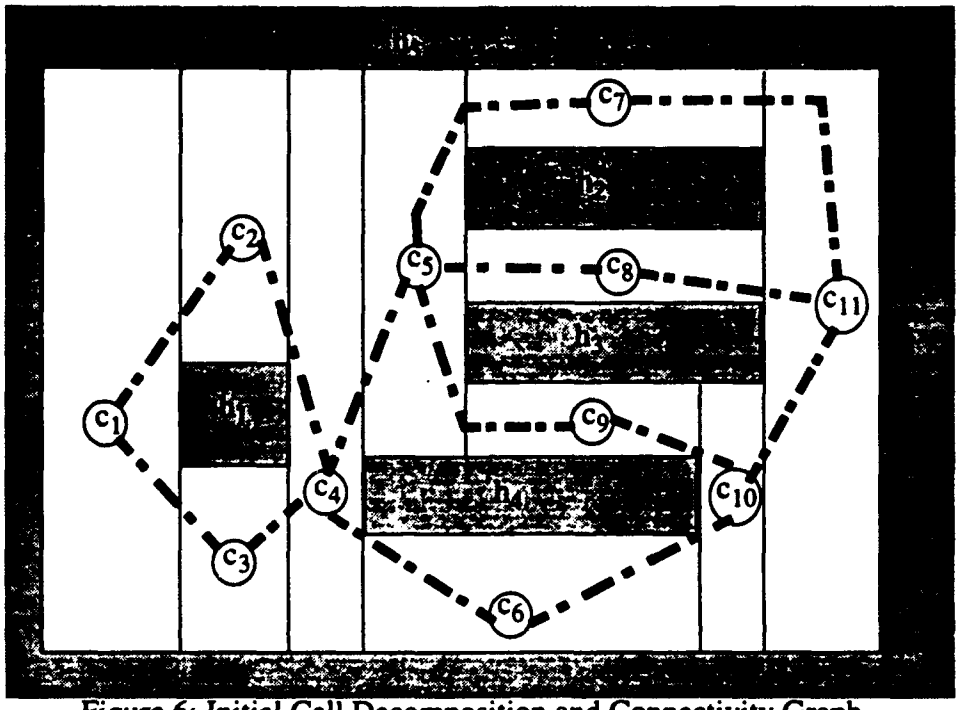

Figure 6: Initial Cell Decomposition and Connectivity Graph

D. AUTOMATIC PATH CLASS GENERATION

We now describe how to use the connectivity graph to automatically generate homotopy classes. It should be obvious that each hole in a world will have at least four fences. As previously discussed, only one fence per hole is required to give the minimal information homotopy classes representation. For consistency, we will always choose this to be the downward fence from the leading extreme vertex. If the hole does not have such a vertex, we will choose the downward fence extending from the leading parallel edge. We will describe the fence and intersecting mode by naming the cell that the robot departs and enters as it crosses the fence. We call this the *cell movement sequence*. For example, from the graph in Figure 6, we see that the cell movement sequence describing the fence and

crossing modes associated with h_1 are c_1-c_3 and c_3-c_1. It should also be obvious that any path from the robot's initial configuration to its goal can be described by a chain of cell movement sequences, c_{init}-c_j-c_k-...-c_l-c_m-c_{goal}, called the *complete cell movement sequence*. Note that the complete cell movement sequence contains an embedded fence crossing sequence, and can therefore be used to represent a homotopy class.

Since any complete cell movement sequence from c_{init} to c_{goal} uniquely defines a homotopy class, we can search the connectivity graph to find all possible paths and thus generate all path classes. We must first, however, define some stopping criteria for the search. Recall that we are only interested in evaluating simple paths. This means that, for most cells in the decomposition, once the robot leaves it should not return. Unfortunately, this is not true for some cells, specifically ambiguous cells. Consider a robot located at the bottom of cell c_6 in Figure 6, on page 24, which needs to move to the top of cell c_4. Suppose the robot is too wide to fit through the gap between h_1 and h_4. If we do not allow the robot to reenter a cell once it departs, the path class given by c_6-c_4-c_3-c_1-c_2-c_4 would not be considered. Clearly, this should be one of the alternatives. At first we might consider relaxing the "visit once" criteria for c_4, but this would lead to consideration of the class given by c_6-c_4-c_2-c_1-c_3-c_4, which clearly is not simple. We must, therefore, eliminate the ambiguity in these cells in order to apply the stopping criteria uniformly.

The first step is to identify ambiguous cells created by the parallel decomposition. Again, the connectivity graph provides a useful tool. From an earlier discussion, we know that cells c_1 and c_{11} are special cases of ambiguous cells, and can be identified immediately without the graph. We can also see that cell c_4 is a standard example of the first form, and c_5 is an example of the second. Additionally, we note that they correspond to nodes of the connectivity graph of degree greater than three. It should be obvious that having degree greater than three is sufficient for being an ambiguous cell by the very nature of their construction. If it is also necessary, then identifying such cells is trivial.

Unfortunately, it is not generally necessary for a cell of degree four or more to be ambiguous. Consider, again, the example in Figure 6. Suppose that holes h_2 and h_3 were connected by the fence between cells c_8 and c_{10}, and were really one large hole. The only change to the connectivity graph would be the removal of the arc from c_8 to c_{10}. Now, cell c_5 still has degree four, but no longer falls under the definition of an ambiguous cell. We would not want any path to enter c_5 more than once. Nevertheless, we can use the connectivity graph to identify candidate cells, and then examine how they were constructed to verify if they are indeed ambiguous.

Since an ambiguous cell is one which does not contain geometric information about L_α's orthogonal axis, it would seem natural to add this information with a supplemental sweep by $L_\beta = L_{\alpha+90}$. We conduct the sweep in much the same way as the initial decomposition, except now we only sweep selected cells. At each extreme vertex defining the ambiguous cell, we extend L_β in both directions until it intersects a hole or a fence. If the cell was defined by one or more parallel edges, then we extend L_β from any point along each edge. For consistency, we will use the midpoint. This divides the original cell into at most n+1 convex pieces, where n is the number of holes which defined the cell. The new convex pieces are called *sub-cells*. Figure 7 shows the results of the supplemental sweep. The new fences are indicated by thin dotted lines.

We must now relabel the sub-cells and reconstruct the connectivity graph. In order to preserve the information contained in the original graph, we will name the sub-cells in the order in which they were created by adding a suffix to their old label. We also want to add sub-cell adjacency arcs while maintaining the original adjacency information. We preserve the original adjacency information by adding only one arc between cells that were adjacent in the original decomposition. For example, c_{5C} is adjacent to both c_{4A} and c_{4B}, yet we only add one arc from c_5 to c_4. It does not matter which arc is added, as long as the sub-cells within these cells are adjacent. Adding all sub-cell adjacency arcs would destroy the bijection between homotopy classes and complete cell movement sequences.

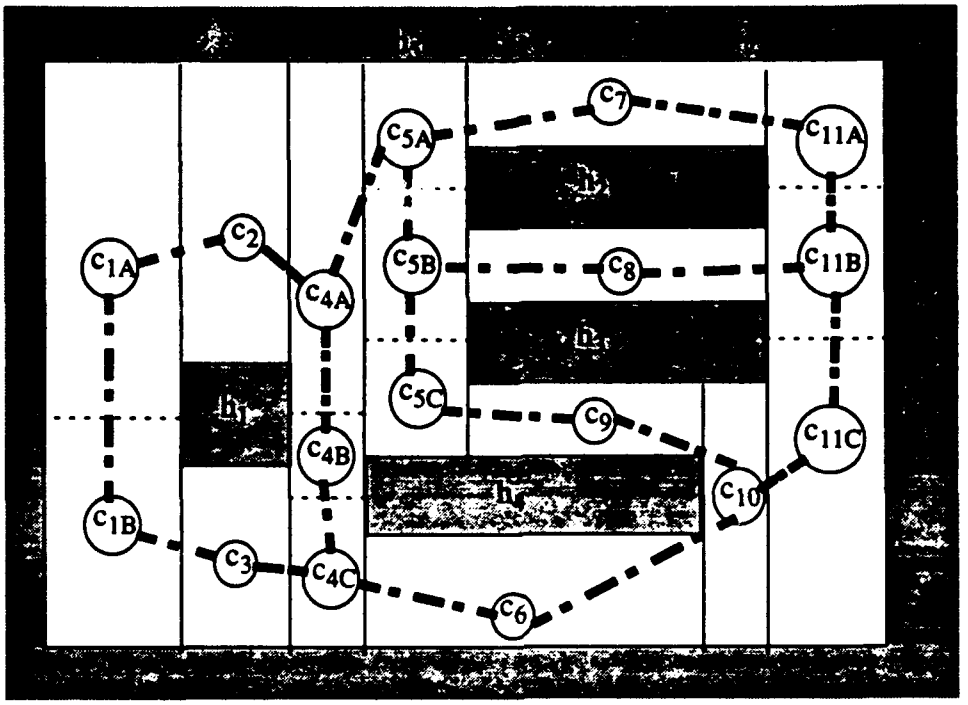

Figure 7: Supplemental Sweep of Ambiguous Cells

Now we see that we can define a simple path to be one that enters a sub-cell or cell only once. A simple path for the robot, then, corresponds to a simple path on the graph. Moreover, to find all simple path classes for the robot, we need only search the graph for all simple paths. We propose using a depth first search starting at c_{init} and terminating at c_{goal}. We can apply a simple backtracking strategy to find all simple paths.

The complete set of path classes is passed to the next layer of the motion planning algorithm, where a class is selected and a detailed motion plan is formed. We present the specific decomposition and graph search implementations in the next chapter.

V. IMPLEMENTATION ON YAMABICO

A. 2-DIMENSIONAL GEOMETRIC MODEL OF A ROBOT'S WORLD

We propose to represent the robot's world by specifying the vertices of the polygonal holes. Each hole, then, becomes an ordered list of vertices such that traversing the list corresponds to traversing the hole's boundary with the free space on the right. In other words, vertices of regular holes are ordered counter-clockwise, while vertices of inverted holes are ordered clockwise. Since information is commonly needed about a vertex's neighbors, the specific data structure must be able to efficiently identify its next and previous vertices. Storing the vertices in a doubly linked list is one alternative. In this chapter we will describe the world of our robot, Yamabico, and provide specific details of how we implement the model and theory discussed earlier.

B. ALGORITHMS AND DATA STRUCTURES

The code for the implementation discussed in this chapter is attached as an appendix. It is also available in the yamabico account under the graduates subdirectory. Currently, building and decomposing the world model is preprocessed on a Unix workstation with the same architecture as Yamabico. We are investigating, however, the possibility of creating these structures using the robot's processor. By processing them on board, we gain the ability to relocate Yamabico without recompiling and redownloading the entire kernel.

1. World Model

Yamabico's world is stored as a circularly linked list of polygons, where each polygon is a doubly linked list of its vertices. Access to the world is gained through a pointer to one of the polygons on the list. The file *build.h* contains the definitions for the actual C structures used, while Figure 8 gives a graphical representation.

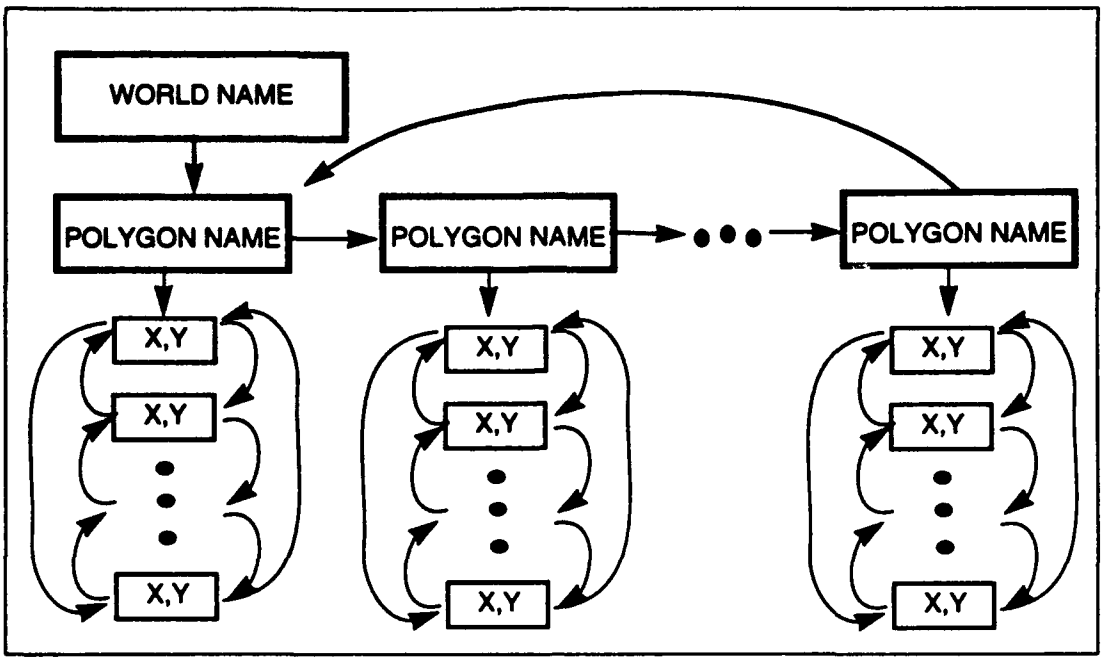

Figure 8: Representation of World Model

Initially, the vertex information is stored as an ASCII file. The function *buildWorld* in the file *build.c* reads the vertices one at a time, and constructs the linked structure described above. This structure is then used for two purposes. The first is to create a file which explicitly defines every polygon and every vertex variable, along with an initialization function to assign them the correct value. This new file is compiled and linked with the robot's kernel. The file *worlddata.c* is an example. The decomposition function also uses the constructed world model to build the cells and connectivity graph. These processes are described below.

2. Cell Decomposition

One of the goals of the data structure used to represent the decomposition of the robot's free space is to answer these two questions efficiently: Whether an edge defines the boundary of a hole, or whether it defines a fence crossing; and in which cell is the robot currently located. Additionally, it is reasonable for the robot's global motion planner to expect that different representations of the free space be consistent. Therefore, we propose

to represent the decomposed world by specifying the vertices of the cells. Specifically, we will model each cell as an inverted hole, with its boundary determined by a combination of obstacles and fences. In this way, any intracell motion planning can be accomplished as if the robot were located in an obstacle-free convex world. Intercell motion is permitted by allowing the robot to cross cell boundaries which are defined by fences. The complete representation of the decomposed world is a circularly linked list consisting only of cells; obstacles are implicitly defined by a subset of the cell boundaries.

The decomposition is achieved by sweeping a vertical line across the original world using the ideas presented in [La91]. Sweeps of orientations other than 90 degrees are accomplished by rotating the world, conducting a vertical sweep, and then rotating the world back to its original configuration. The continuous sweep is discretized by realizing that cells are created or completed only at obstacle vertices, and that they are only affected by obstacle edges which have non-empty intersection with the sweep line. This immediately defines the need for two data structures: a list of *events*, which are the vertices ordered by x-coordinate, and a list of those edges which intersect the sweep line. The event list is static for a given world, but the edge list changes at each event. We can, therefore, create the event list as we read in the initial world data. Events are maintained as pointers to vertices, and are properly placed on the event list using a simple linear insertion. The sweep is performed by the algorithms shown in Figures 9 through 13.

We refer the reader to the actual code for the precise implementation details, but make some explanatory notes here. For simplicity, we show the main algorithm assuming that the sweep line only encounters one event at a time. The actual program handles the general case where the sweep line may encounter multiple events simultaneously. Also, the active edge list is maintained so that edges are ordered by decreasing y-coordinate of their intersection with the sweep line. This allows us to quickly locate the edge which first intersects the fence as it is extended from the current event vertex. These algorithms are implemented in the files *decompose.c* and *decomposutil.c*.

```
ALGORITHM DECOMPOSE WORLD
INPUT: Event list sorted by x-coordinate; Model of robot's world
OUTPUT: Model of robot's world represented as convex cells

begin

    while event list is not empty
        currentEvent ← next Event from Event List

        ADD EDGES TO ACTIVE LIST;

        FINISH COMPLETED CELLS;

        START NEW CELLS;

        REMOVE EDGES FROM ACTIVE LIST;

    end while;
end DECOMPOSE WORLD
```

Figure 9: Algorithm DECOMPOSE WORLD

```
ALGORITHM ADD EDGES TO ACTIVE LIST
INPUT: Event
OUTPUT: Updated Active Edge List

begin

    currentVertex ← vertex pointed to by currentEvent
    if x-coordinate of currentVertex's next vertex > x-coordinate of currentVertex then
        add edge defined by currentVertex and currentVertex's next vertex
        to Active Edge List in the proper order

    if x-coordinate of currentVertex's previous vertex > x-coordinate of currentVertex
    then
        add edge defined by currentVertex and currentVertex's previous vertex
        to Active Edge List in the proper order

end ADD EDGES TO ACTIVE LIST
```

Figure 10: Algorithm ADD EDGES TO ACTIVE LIST

```
ALGORITHM COMPLETE CELL
INPUT: Event
OUTPUT: Complete cell added to world model, Updated Cell in Progress List

begin

    currentVertex ← vertex pointed to by currentEvent
    if currentVertex is convex minimal extreme
        complete cell UP/DOWN
    elsif currentVertex is convex maximal extreme
        complete cell UP
        complete cell DOWN
    elsif currentVertex is convex interior up
        complete cell UP
    elsif currentVertex is convex interior down
        complete cell DOWN
    elsif currentVertex is concave maximal extreme
        complete ISOLATED cell
    elsif currentVertex is concave interior
        extend cell

end COMPLETE CELL
```

Figure 11: Algorithm COMPLETE CELL

```
ALGORITHM START NEW CELL
INPUT: Event
OUTPUT: Updated Cells in Progress List

begin

    currentVertex ← vertex pointed to by currentEvent
    if currentVertex is convex minimal extreme
        start new cell UP
        start new cell DOWN
    elsif currentVertex is convex maximal extreme
        start new cell UP/DOWN
    elsif currentVertex is convex interior up
        start new cell UP
    elsif currentVertex is convex interior down
        start new cell DOWN
    elsif currentVertex is concave minimal extreme
        star new ISOLATED cell

end START NEW CELL
```

Figure 12: Algorithm START NEW CELL

```
ALGORITHM DELETE EDGES FROM ACTIVE LIST
INPUT: Event
OUTPUT: Updated Active Edge List

begin

    currentVertex ← vertex pointed to by currentEvent
    for each Edge on Active Edge List

        if x-coordinate of currentVertex = x-coordinate of Edge's trailing vertex then
            remove edge from Active Edge List

    end for

end DELETE EDGES FROM ACTIVE LIST
```

Figure 13: Algorithm DELETE EDGES FROM ACTIVE LIST

3. Connectivity Graph

We generate the connectivity graph as a by-product of the decomposition sweep. The graph is maintained as an adjacency list. Nodes of the graph are added when a cell is started. As a cell is completed, the vertical fence which defines its rightmost boundary is placed on a temporary holding list. Then, as the algorithm enters the start new cell phase, the leftmost boundary of each new cell is compared with the boundaries on the holding list. Two boundaries will match if the cells are adjacent. Once adjacency of two cells is determined, both nodes on the graph are updated.

4. Determining Cell of Current and Goal Configurations

If the initial and goal configurations are known before the world is decomposed, the cell in which they are located can be determined as part of the sweep. Since the world is presently decomposed off line, however, we will assume that neither location is known. The first step in identifying the homotopy classes, then, is to identify the cell containing the robot's initial position and the cell containing the goal. Once the robot has this information,

it should be able to keep its current cell location updated by using odometry control. Still, it may be necessary for the robot to verify this information.

Preparata and Shamos [PrSh85] present two algorithms for determining whether a point lies within the interior of a simple N-gon. Including preprocessing, each takes O(N) time. Since we are concerned with only convex cells, we have adapted the more restrictive convex inclusion algorithm. It is based on the fact that for any point p interior to a convex polygon, and for any two vertices v_1 and v_2 of that polygon, the angle formed by the two rays $\overline{pv_1}$ and $\overline{pv_2}$ is positive, if v_2 is reached from v_1 while traveling the polygon's border in its natural direction. Now, if for each vertex v_1 and its next vertex v_2, this angle is positive, we know that p lies within the polygon. We can use the algorithm in Figure 14, to find the cell of the initial and goal configurations, and the algorithm in Figure 15, to verify that the robot is located within a specific cell.

```
ALGORITHM FIND CELL
INPUT: Configuration, Decomposed Model of robot's world
OUTPUT: Cell which contains Configuration

begin

   while all cells not checked
      currentCell ← next cell not checked

      if INSIDE CELL (currentCell)
         Configuration is located within currentCell

   end while

   ERROR: Configuration is not located within robot's world

end FIND CELL
```

Figure 14: Algorithm FIND CELL

```
ALGORITHM INSIDE CELL
INPUT: Configuration, Convex cell
OUTPUT: TRUE if Configuration is within Convex Cell, FALSE otherwise

begin

    while all vertices not checked
        currentVertex ← currentVertex's next vertex

        if Configuration, currentVertex, and currentVertex's next Vertex make a right turn
            continue;
        else
            return FALSE;
        end if

    end while

    return TRUE;

end INSIDE CELL
```

Figure 15: Algorithm INSIDE CELL

5. Finding All Homotopy Classes

Once we know the cell containing the robot's configuration, and the cell containing the goal configuration, we can apply a depth-first search with backtracking to the connectivity graph to find all simply homotopy classes. We will use the algorithm in Figures 16 and 17, to find all classes, and to give their complete cell movement sequence representation.

We mention, here, an alternative to searching the graph for all homotopy classes, but we neither develop it in this thesis nor presently implement it on Yamabico. We begin by applying edge weights to the graph using a criteria based on the cost of moving from one cell to the other. Then, we can use Dijkastra's algorithm to search the graph for the path of minimal cost. Two possible cost factors to consider are the length of the path segment, and the clearance between obstacles. Unfortunately, it appears to be a very difficult

problem to find the exact cost for a cell movement sequence. We could use, instead, a good approximation to find a near-optimal solution.

```
ALGORITHM FIND PATHS
INPUT: Connectivity Graph, Cell of Goal Configuration, Cell of Robot's Configuration
OUTPUT: Complete cell movement sequence of each homotopy class

begin

    if Robot and Goal are in the same cell
        return NULL path class

    initialize all predecessors to NILL
    CG ← Connectivity Graph
    s ← Cell of Robot's Configuration
    g ← Cell of Goal Configuration
    DFS(CG, s, g)

end FIND PATHS
```

Figure 16: Algorithm FIND PATHS

```
ALGORITHM DFS
INPUT: Connectivity Graph, u, v vertices in Connectivity Graph
OUTPUT: Complete cell movement sequence of a homotopy class

begin

    if u = v
        return path class by tracing predecessors
    else
        for all vertices x which are adjacent to u do

            if predecessor of x = NIL
                mark predecessor of x = u
                DFS(CG,x,v)
                mark predecessor of x = NIL

end FIND PATHS
```

Figure 17: Algorithm DFS

C. INTEGRATION WITH MML

Yamabico's global motion planner has been mentioned several times in this chapter. Currently, the complete system is under development by Chien-Liang Chuang and Joseph Kovalchick, PhD candidates at the Naval Postgraduate School. When completed, it will provide the main interface as part of the model-based mobile robot language (mml). The work done in this thesis will primarily be a subset of the inner workings of the global motion planner. We can, however, provide some limited integration now.

We hide the details of the world model by providing access through a pointer to the world structure. The world structure, then, is linked to one of the polygons. Multiple representations of the same world can be present simultaneously. Each, though, are independent and share no information. Presently, the only interface to the world models is through the configuration-cell location functions. As the global motion planner evolves, more interface functions must be provided.

VI. CONCLUSION

A. RESULTS

Without the global motion planner, it is hard to test and validate the theory described in this thesis. Fortunately, that work is underway. Still, we believe that the ideas and implementation presented here will provide a solid framework for future work. We have been able to graphically analyze the decomposition process, and evaluate the generation of the connectivity graph. The functions that will provide the initial interface between the global planner and the robot's world model have all been tested on board Yamabico. The results are a reliable first layer to what will be a robust, multi-layer, autonomous motion planner.

B. AREAS OF FURTHER RESEARCH

While the parallel cell decomposition provides a good beginning for the global motion planner, it has left some questions unanswered, and raised others. We present three of them here; two relating to the theory of our layered path planning paradigm, and one relating to the implementation on Yamabico.

1. Orientation of L_α

We mentioned previously that the generation of some ambiguous cells could be avoided by carefully choosing L_α. Furthermore, some worlds may be better suited for a sweep by one orientation over another. This leads us to the question of whether we can efficiently determine if one orientation of L_α is better than another for a given world. Consider the world and the two decompositions in Figure 18, on page 40, which is similar to a portion of the 5th floor of Spanagel Hall. We notice that the horizontal decomposition creates many more cells than the vertical decomposition. On the one hand, we end up further decomposing many of the classrooms which were already convex. On the other hand, though, we get many more motion clues as we move along the long hallway. It seems apparent that the two decompositions shown are better than any other, but we do not know

for sure. Also, how can we decide if the additional information provided in the hallway is worth the extraneous information added to the classrooms. We leave this as an open question, but use it to provide, perhaps, some insight into a solution to another problem.

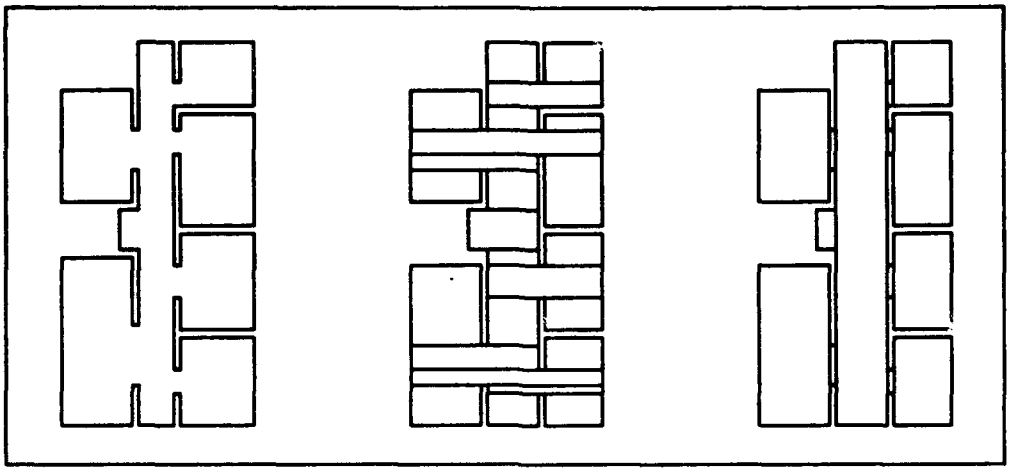

Figure 18: A robot's world and two decompositions

2. Ambiguous Cells

While the generation of some ambiguous cells can be avoided with a particular choice of L_α, we realize that generally some will still be created. This is inherent in the fact that for a given sweep, we only preserve spatial information in one orientation. The method we proposed of limiting a supplemental orthogonal sweep to the ambiguous cells is acceptable, but this still raises some questions. Primarily, we are uneasy with the idea that this method favors one orientation over another. We investigated the possibility of fully decomposing the world using a set of orthogonal sweeps, and generating a corresponding set of connectivity graphs. The idea was to independently find all homotopy classes with both graphs, and provide the lower layers of the global motion planner a pair of cell movement sequences for each class. This approach has two benefits. First, we end up treating both orientations equally. Second, we provide additional motion clues that would be missing from a single sweep. In the example from Figure 18, the second sweep would

allow the global motion planner to use the additional cells in the hallway, while disregarding them in the classrooms.

Unfortunately, we encountered a problem with this approach. For a world with multiple homotopy classes, we are still unsure how to accurately make the pairings. It is easy for a human to match a simple path on one graph with a simple path on the other, but as of yet, we have not found a method to do it autonomously.

3. Downloading the World Model

Currently, the model for the robot's world and the decomposition are transformed into C files, compiled, and linked into the robot's kernel. This method will cause us to relink and redownload the kernel every time we want to change the robot's world. A better solution would be to allow Yamabico to build and decompose the world on board. Then, it needs only download the new vertex information when relocated. We are presently investigating a method for storing the ASCII world information in RAM onboard Yamabico, which will allow us to move the construction and decomposition from the workstation to the robot.

LIST OF REFERENCES

[AlWe89] Alt, H., and Welzl, E., Visibility Graphs and Obstacle-Avoiding Shortest Paths, *Zeitschrift fur Operations Research, Volume 32, pages 145-164*, 1989.

[An92] Andreae, Thomas, Some Results on Visibility Graphs, *Discrete Applied Mathematics, Volume 40*, 1992.

[Chz87] Chazelle, B. Approximation and Decomposition of Shapes, in [ScYa87], 1987.

[ChKe90] Chew, L. Paul, and Kedem, Klara, *High-Clearance Motion Planning for a Convex Polygon among Polygonal Obstacles*, Department of Computer Science, Cornell University Technical Report, 1990.

[Cr78] Croom, Fred H., *Basic Concepts of Algebraic Topology*, Springer-Verlag, 1978.

[GaGr83] Gamelin, Theodore W., and Greene, Robert Everist, *Introduction to Topology*, Saunders College Publishing, 1983.

[Je91] Jenkins, Kevin D., *The Shortest Path Problem in the Plane with Obstacles: A Graph Modeling Approach to Producing Finite Search Lists of Homotopy Classes*, Masters Thesis, Naval Postgraduate School, 1991.

[Ka94] Kanayama, Yutaka, *Mathematical Theory of Robotics: Inrtoduction to 2D Spatial Reasoning*, Advanced Robotics Class Notes, Naval Postgraduate School, 1994.

[Ki89] Kirkwood, James R., *An Introduction to Analysis*, PWS-KENT Publishing Company, 1989.

[Kl89] Klein, R. Concrete and Abstract Voronoi Diagrams, *Lecture Notes in Computer Science, Vol 400*, Springer-Verlag, 1989.

[La91] Latombe, Jean-Claude, *Robot Motion Planning*, Kluwer Academic Publishers, 1991.

[Lee78] Lee D.T., *Proximity and reachability in the plane*, PhD Dissertaion, University of Illinois at Urbana-Champaign, 1978.

[LMW87] Luccio, F., Mazzone, S., Wong, C.K., A Note on Visibility Graphs, *Discrete Mathematics, Volume 64*, 1987.

[MMO91] Mehlhorn, K. Meiser S., O'Dúnlaing, C. On the Construction of Abstract Voronoi Diagrams, *Discrete and Computational Geometry 6(1991)*, Springer-Verlag, 1991.

[OR87] O'Rourke, Josheph, *Art Gallery Theorems and Algorithms*, Oxford University Press, 1987.

[PrSh85] Preparata, Franco P., and Shamos, Michael Ian, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[ScYa87] Schwartz, J.T., and Yap, C.K., *Algoithmic and Geometric Aspects of Robotics*, 1987.

[ShHo75] Shamos, M. I., and Hoey, D., Closest - Point Problems, *Proceedings of the 16th Annual Symposium of FOCS*, 1975.

[Yama93] Yamabico Research Group, *Users Guide to Yamabico*, Naval Postgraduate School, 1993.

[Yama94] Yamabico Research Group, *Users Guide to MML*, Naval Postgraduate School, 1994.

APPENDIX A

A. MAIN

1. main.c

```
/*****************************************************************
FILE: main.c
PURPOSE: This file contains the main function which parses the command
        line and then calls the decompose functions
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "util.h"
#include "build.h"
#include "decompose.h"
#include "worldfile.h"

int
main(int argc, char* argv[])
{

   FILE* plotFile;
   FILE* worldFile;

   world* testWorld;
   world* decompWorld;
   polygon* curPolygon;
   vertex* curVertex;
   event* eventList;
   event* currentEvent;

   node* cGraph;

   int numberOfSweeps;
   int counter;
   double sweepAngle;
   double cosRotInv;
   double sinRotInv;
   double rotX, rotY;

   char worldTag[7];
   char filename[21];

   eventList = NULL;
   cGraph = NULL;
```

```
  decompWorld = NULL;

  /* This block of code parses the command line to ensure it is correct*/
  if (argc < 2){
     printf("\nYou must supply the name of the file containing\n");
     printf("the vertices of the robot's world\n");
     return -1;
  }else if (argc < 3){
     numberOfSweeps = 1;
     sweepAngle = 90.0;
  }else if ((numberOfSweeps = atoi(argv[2])) > 0){
     if (argc != (numberOfSweeps + 3)){
        printf("\n%d sweeps specified, but %d angles given\n",
              numberOfSweeps,(argc - 3));
        return -4;
     }
     for (counter = 1; counter <= numberOfSweeps; counter++){
        sweepAngle = atof(argv[2+counter]);
        if ((sweepAngle <= 0.0) || (sweepAngle > 180.0)){
           printf("\n %Sweep angle must be in the range (0.0-180.0]\n");
           return -2;
        }
     }
  }else{
     printf("\nInvalid commandline options. Correct format is:\n");
     printf("DecomposeWorld filename [# of sweeps] [sweep angles]\n");
     return -3;
  }

  /* read in the world datafile, build the structures for the
    original world model, and create the outputfile */

  if((testWorld = buildWorld(argv[1], 90, &eventList)) == NULL){
     return -3; /*file does not exist, or contains an error*/
  }else{
     createWorldFile(testWorld,NULL, "originalworld.c",90,0,"0");
  }
  /* Decompose the world for every sweep angle specified as a
    command line parameter*/

  for (counter = 1; counter <= numberOfSweeps; counter++){
     if (argc>2)
        sweepAngle = atof(argv[2+counter]);
     sprintf(worldTag,"%3.2f",sweepAngle);
     (*strchr(worldTag,'.'))='_';
     freeWorld(&testWorld,&decompWorld,&eventList,&cGraph);
     testWorld = buildWorld(argv[1],sweepAngle, &eventList);
     decompWorld = decompose(eventList, &cGraph);
     sprintf(filename,"decompworld_%s.c",worldTag);
  createWorldFile(decompWorld,cGraph,filename,sweepAngle,counter,worldTag);
}
```

```
    /* If only one decomposition was specified, create a file which can
       be used by GnuPlot to graphically show the cells */
    if (argc <= 4){
        cosRotInv = cos((-(M_PI*(90.0-sweepAngle)))/180.0);
        sinRotInv = sin((-(M_PI*(90.0-sweepAngle)))/180.0);

       curPolygon = decompWorld ->polygonList;
       plotFile = fopen("plot.dat","w");
       do{
          fprintf(plotFile,"#Cell %s vertices\n",curPolygon->name);
          curVertex = curPolygon->vertexList;
          do{
             /*****"re-rotate" the vertices******/
             rotX = ((curVertex->posit.X*cosRotInv)-(curVertex->posit.Y*sinRotInv));
             rotY = ((curVertex->posit.X*sinRotInv)+(curVertex->posit.Y*cosRotInv));
             fprintf(plotFile,"%4.2f %4.2f\n", rotX, rotY);
             curVertex = curVertex->next;
          }while (curVertex!= curPolygon->vertexList);
             rotX = ((curVertex->posit.X*cosRotInv)-(curVertex->posit.Y*sinRotInv));
             rotY = ((curVertex->posit.X*sinRotInv)+(curVertex->posit.Y*cosRotInv));
             fprintf(plotFile,"%4.2f %4.2f \n\n", rotX, rotY);
          curPolygon = curPolygon->next;
       }while (curPolygon != decompWorld->polygonList);
       fclose(plotFile);
       printf("Your plot data is located in plot.dat\n" );
    }

    return 1;

}
```

B. BUILD

1. build.h

```
/*********************************************************************
FILE: build.h
PURPOSE: This file contains the definitions for the types used in
        the world model.
*********************************************************************/

#ifndef __build_h
#define __build_h


typedef struct point{
   double X;
   double Y;
}point;

typedef struct vertex{
```

```
  point posit;
  char bndry[5];
  struct vertex* next;
  struct vertex* prev;
}vertex;

typedef struct polygon{
  char name[5];
  int mode;
  vertex* vertexList;
  vertex* open1;  /*used while constructing cells*/
  vertex* open2;  /*used while constructing cells*/
  struct polygon* next;
}polygon;

typedef struct world{
  char name[15];
  polygon* polygonList;
}world;

typedef struct event{
  vertex* eVertex;
  struct event* next;
  polygon* owner;
}event;

typedef struct arc{
    struct node* Node;
    struct arc* next;
    double weight;
    int visited;
}arc;

typedef struct node{
    polygon* cell;
    arc* arcList;
    struct node* predecessor;
    struct node* next;
    arc* curArc;
}node;

/*****************buildWorld(char*, double, event**)******************
  This function reads the ASCII vertex information in the inputfile,
  rotates the world by decompostion angle, and orders the
  vertices to create the event list. It returns a pointer
  to the world.
*********************************************************************/
world* buildWorld(char*, double, event**);

#endif
```

2. build.c

```
/*****************************************************************
FILE: build.c
PURPOSE: This file contains the function which reads in the ASCII
      vertex information and creates the linked list structure
      defined by the world model.
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include "util.h"
#include "build.h"
#include "dec mpose.h"


world*
buildWorld(char* fileName, double rot, event** eventList)
{

  FILE* inputFile;
  char ch;

  world* currentWorld;
  polygon* currentPolygon;
  vertex* currentVertex;
  vertex* previousVertex;

  double tempX;
  double tempY;
  double cosRot;
  double sinRot;

  event* newEvent;
  event* currentEvent;
  event* previousEvent;

  /*We rotate the world to achieve sweeps other than vertical, here
    we calculate the trig functions for the rotation transformation*/
  rot = (M_PI * (90.0-rot))/180.0;
  cosRot = cos(rot);
  sinRot = sin(rot);

  if ((inputFile = fopen(fileName, "r")) == NULL){
     printf("\nCould not open named file: %s\n",fileName);
     return NULL;
  }

  currentWorld = (world*)malloc(sizeof(world));
```

```
currentPolygon = currentWorld->polygonList =
        (polygon*)malloc(sizeof(polygon));

getName(inputFile, currentWorld->name, 15);

while(1){/*loop until all polygons are read*/

  getName(inputFile, currentPolygon->name,5);
  fscanf(inputFile,"%d(",&currentPolygon->mode);

  currentVertex = currentPolygon->vertexList =
        (vertex*)malloc(sizeof(vertex));

  while(1){/*loop until all vertices for this polygon are read*/

    fscanf(inputFile,"%lf,%lf)",&tempX, &tempY);
    /*rotate vertices of world depending on sweep angle*/
    (currentVertex->posit.X)= ((tempX*cosRot)-(tempY*sinRot));
    (currentVertex->posit.Y)= ((tempX*sinRot)+(tempY*cosRot));

    /*building event list*/
    if ((*eventList) == NULL){
      (*eventList) = (event*)malloc(sizeof(event));
      (*eventList)->eVertex = currentVertex;
      (*eventList)->owner = currentPolygon;
      (*eventList)->next = NULL;
    }else{

      newEvent = (event*)malloc(sizeof(event));
      currentEvent = (*eventList);
      previousEvent = (*eventList);
      while ((currentEvent) &&
          ((currentVertex->posit.X > currentEvent->eVertex->posit.X) ||
          ((currentVertex->posit.X == currentEvent->eVertex->posit.X) &&
          (currentVertex->posit.Y < currentEvent->eVertex->posit.Y)))){
        previousEvent = currentEvent;
        currentEvent = currentEvent->next;}

      newEvent->next = currentEvent;
      if(currentEvent == (*eventList))
        (*eventList) = newEvent;
      else
        previousEvent->next = newEvent;

      newEvent->eVertex=currentVertex;
      newEvent->owner = currentPolygon;
    }

    /*end building event list */

    do
```

```
            ch = getc(inputFile);
          while(isspace(ch));
          if (ch == ')')
            break; /*last vertex read for this polygon*/
          else if (ch != '('){
            fclose(inputFile);
            fprintf("\nError in vertex data file\n");
            return NULL;
          }else{ /*read in left paren of next vertex*/
            currentVertex->next=(vertex*)malloc(sizeof(vertex));
            previousVertex = currentVertex;
            currentVertex = currentVertex->next;
            currentVertex->prev = previousVertex;
          }

        }/*end vertex while loop*/

        currentVertex->next = currentPolygon->vertexList;
        currentPolygon->vertexList->prev = currentVertex;

        do
          ch = getc(inputFile);
        while(isspace(ch));
        if (ch == ')'){/*last polygon read*/
          currentPolygon->next = currentWorld->polygonList;
          fclose(inputFile);
          return currentWorld;
        }else {/* read in first character of next polygon name */
          ungetc(ch,inputFile);
          currentPolygon->next = (polygon*)malloc(sizeof(polygon));
          currentPolygon = currentPolygon->next;
        }
      }/*end polygon while loop*/
    }
```

C. DECOMPOSE

1. decompose.h

```
/*************************************************************************
FILE: decompose.h
PURPOSE: This file contains the prototype for the function which
      examines the the vertices of the world and determines what
       action to take in the following areas:
                adding edges, deleting edges, starting cells,
                and finishing cells.
*************************************************************************/

#ifndef __decompose_h
#define __decompose_h
```

```
#include "build.h"

world* decompose( event*, node**);

#endif
```

2. decompose.c

```
/*************************************************************************
FILE: decompose.c
PURPOSE: This file contains the function which examines the the vertices
      of the world and determines what action to take in the
      following areas:  adding edges, deleting edges, starting cells,
                  and finishing cells.
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "build.h"
#include "decompose.h"
#include "util.h"
#include "decomputil.h"


int cellCount;

world*
decompose(event* eventList, node** cGraph)
{

   world* robotsWorld;

   edge* activeEdges;
   edge* cEdge;
   event* currentEvent;
   event* simulEvent;
   polygon* cellList;
   fence* activeFences;


   robotsWorld = malloc(sizeof(world));
   robotsWorld->polygonList = NULL;
   activeEdges = NULL;
   cellList = NULL;
   cellCount = 0;
   activeEdges = NULL;
   activeFences = NULL;
   currentEvent = eventList;
```

```
while(currentEvent){

  /*******ADD EDGES LOOP*********/
  simulEvent = currentEvent;
  do{
    addEdge(simulEvent, &activeEdges);
    simulEvent = simulEvent->next;
  }while((simulEvent) &&
    (currentEvent->eVertex->posit.X == simulEvent->eVertex->posit.X));

  /*******FINISH CELLS LOOP*******/
  simulEvent = currentEvent;
  do{
    if (simulEvent->eVertex->bndry[0]=='V')/*convex vertex*/
    {

      if ((simulEvent->eVertex->posit.X > simulEvent->eVertex->next->posit.X)
       &&(simulEvent->eVertex->posit.X <= simulEvent->eVertex->prev->posit.X))
        /* this is an up interior convex vertex*/
      {
        if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
          finishCellUp(simulEvent, &cellList,
              activeEdges,&robotsWorld, &activeFences);

      }else
      if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
       &&(simulEvent->eVertex->posit.X > simulEvent->eVertex->prev->posit.X))
        /* this is a down exterior convex vertex*/
      {
        finishCellDown(simulEvent, &cellList,
            activeEdges,&robotsWorld,&activeFences, 0);

      }else
      if ((simulEvent->eVertex->posit.X <= simulEvent->eVertex->next->posit.X)
       &&(simulEvent->eVertex->posit.X < simulEvent->eVertex->prev->posit.X))

      {
        /* this is a minimal exterior convex  vertex*/
        if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
        finishCellUpDown(simulEvent, &cellList,
            activeEdges,&robotsWorld,&activeFences);

      }else
      if ((simulEvent->eVertex->posit.X >= simulEvent->eVertex->next->posit.X)
       &&(simulEvent->eVertex->posit.X > simulEvent->eVertex->prev->posit.X))
      {
        /* this is a maximal exterior convex vertex*/
        if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
          finishCellUp(simulEvent, &cellList,
            activeEdges,&robotsWorld, &activeFences);
        finishCellDown(simulEvent, &cellList,
```

```
                activeEdges,&robotsWorld, &activeFences, 0);

        }
    }else{/* concave vertex*/
        if ((simulEvent->eVertex->posit.X > simulEvent->eVertex->next->posit.X)
        &&(simulEvent->eVertex->posit.X >= simulEvent->eVertex->prev->posit.X))
        {
            /* this is a maximal extreme concave vertex*/
            if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
                finishCellIsolated(simulEvent, &cellList, &robotsWorld);

        }else
        if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
        &&(simulEvent->eVertex->posit.X > simulEvent->eVertex->prev->posit.X))
        {
            /* this is an interior concave vertex*/
            if ((simulEvent->eVertex->bndry[1]=='O') && (cellList))
                extendCell(simulEvent, &cellList);/*only vertex*/
        }else
        if ((simulEvent->eVertex->posit.X == simulEvent->eVertex->next->posit.X)
        &&(simulEvent->eVertex->posit.X > simulEvent->eVertex->prev->posit.X)){
            /* this is a lower right corner*/
            finishCellDown(simulEvent, &cellList, activeEdges,
                    &robotsWorld, &activeFences, 1);

        }
    }
    simulEvent = simulEvent->next;
}while((simulEvent) &&
    (currentEvent->eVertex->posit.X == simulEvent->eVertex->posit.X));

simulEvent = currentEvent;

/***************START CELLS LOOP **************/
do{
    if (simulEvent->eVertex->bndry[0]=='V')/*convex vertex*/
    {


        if ((simulEvent->eVertex->posit.X >= simulEvent->eVertex->next->posit.X)
        &&(simulEvent->eVertex->posit.X < simulEvent->eVertex->prev->posit.X))
        /*this is an up interior convex vertex*/
        {
            if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
                startCellUp(simulEvent, &cellList, activeEdges,
                        &activeFences, cGraph);

        }else
        if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
        &&(simulEvent->eVertex->posit.X >= simulEvent->eVertex->prev->posit.X))
        /* this is a down exterior convex vertex*/
```

```
    {
      startCellDown(simulEvent, &cellList,
            activeEdges, &activeFences, cGraph);

    }else
    if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
     &&(simulEvent->eVertex->posit.X < simulEvent->eVertex->prev->posit.X))
    {
     /* this is an minimal exterior convex vertex*/

      if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
        startCellUp(simulEvent, &cellList,
          activeEdges, &activeFences, cGraph);
      startCellDown(simulEvent, &cellList,
            activeEdges, &activeFences, cGraph);

    }else{

     /*this is a maximal exterior convex vertex*/
      if (simulEvent->eVertex->bndry[1]=='O')/*only vertex*/
        startCellUpDown(simulEvent, &cellList,
            activeEdges, &activeFences, cGraph);

    }
  }else{/*concave vertex*/
    if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
     &&(simulEvent->eVertex->posit.X < simulEvent->eVertex->prev->posit.X))
    {
     /* this is a minimal extreme concave vertex*/
      startCellIsolated(simulEvent, &cellList, cGraph);

    }else
    if ((simulEvent->eVertex->posit.X >= simulEvent->eVertex->next->posit.X)
     &&(simulEvent->eVertex->posit.X < simulEvent->eVertex->prev->posit.X))
    {
     /* this is a maximal extreme concave vertex*/
      if ((simulEvent->eVertex->bndry[1]=='O') && (cellList))
        extendCell(simulEvent, &cellList);/*only vertex*/

    }else
    if ((simulEvent->eVertex->posit.X < simulEvent->eVertex->next->posit.X)
     &&(simulEvent->eVertex->posit.X == simulEvent->eVertex->prev->posit.X))
     {
        /* this is an upper right corner */
        startCellDown(simulEvent, &cellList,
            activeEdges, &activeFences, cGraph);
     }
  }
  simulEvent = simulEvent->next;
}while((simulEvent) &&
  (currentEvent->eVertex->posit.X == simulEvent->eVertex->posit.X));
```

```
        /***********DELETE EDGES LOOP ***************/
        simulEvent = currentEvent;
        do{
          deleteEdge(simulEvent, &activeEdges);
          simulEvent = simulEvent->next;


        }while((simulEvent) &&
          (currentEvent->eVertex->posit.X == simulEvent->eVertex->posit.X) &&
          (activeEdges));
        currentEvent = simulEvent;


      }


      /*The decomposition is complete, link up the last and first polygons*/
      cellList=robotsWorld->polygonList;
        while(cellList->next)
          cellList = cellList->next;
      cellList->next = robotsWorld->polygonList;

      return robotsWorld;
    }
```

D. DECOMPOSE UTILITIES

1. decomputil.h

```
/*********************************************************************
FILE: decomposutil.h
PURPOSE: This file contains the prototypes for the functions which
      do the actual work for the decomposition.

*********************************************************************/


#ifndef __dcmputil_h
#define __dcmputil_h

#include "build.h"

typedef struct edge{
 vertex* leadingVertex;
 vertex* trailingVertex;
 double xPoint;
 struct edge* next;
}edge;

typedef struct fence{
```

```
    vertex* topVertex;
    vertex* bottomVertex;
    polygon* owner;
    struct fence* next;
}fence;

/**********intersect(edge*, vertex*)**************************
This function computes the y-coordinate of the intersection of the
extension of the sweep line and the current edge. The x-coordinate is
the same as the current event.

*****************************************************************/
double intersect(edge* , vertex*);

/**********addEdge(event*, edge**)*****************************
This function adds edges to the active edge list

*****************************************************************/
int addEdge(event*, edge**);

/**********deleteEdge(event* edge**)***********************
This function deletes edges from the active edge list

*****************************************************************/
int deleteEdge(event*, edge**);

/*********startCellUp(event*, polygon**, edge*, fence**, node*)********
This function finds the vertices for the start of an up cell

*****************************************************************/
void startCellUp(event*, polygon**, edge*, fence**, node**);

/*********startCellDown(event*, polygon*, edge*, fence**, node**)*******
This function finds the vertices for the start of a down cell

*****************************************************************/
void startCellDown(event*, polygon**, edge*, fence**, node**);

/*********startCellUpDown(event*, polygon*, edge*, fence*, node**)********
This function finds the vertices for the start of an up/down cell

*****************************************************************/
void startCellUpDown(event*, polygon**, edge*, fence**, node**);

/***********startCellIsolated(event*, polygon**, node**)**************
ADD ISOLATED CELL TO WORKING LIST
*****************************************************************/
void startCellIsolated(event*, polygon**, node**);

/*****finishCellUp(event*, polygon**, edge*, world**, fence**)***************
This function locates which cell on the active list is completed by this
```

```
event, adds the apprpriate vertices, and adds new cell to the world

**********/
void finishCellUp(event*, polygon**, edge*, world**, fence**);

/******finishCellDown(event*, polygon**, edge*, world**, fence**, int)******
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

**********/
void finishCellDown(event*, polygon**, edge*, world**, fence**, int);

/*****finishCellUpDown(event*, polygon**, edge*, world**, fence*)*******
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

**********/
void finishCellUpDown(event*, polygon**, edge*, world**, fence**);

/*****finishCellIsolated(event*, polygon**, world**)******************
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

**********/
void finishCellIsolated(event*, polygon**, world**);

/************extendCell(event* polygon**)*****************************
This function locates which cell on the active list needs to be
extended by this concave vertex

**********/
void extendCell(event*, polygon**);

/******putFence(fence**, polygon*, vertex*, vertex*)*******************
This function puts an adjacency fence on a list to be matched later

**********/
void putFence(fence**, polygon*, vertex*, vertex*);

/**********updateAdj(fence**, polygon*, vertex*, vertex*, node**)*********
This function matches an adjacency fence with one already on the list
to determine cell adjacency, and then updates the adjacency graph

**********/
void updateAdj(fence**,polygon*, vertex*, vertex*,node**);

/**********************addNode(node**, polygon**)**********************
This function adds a node to the connectivity graph whenever a cell
is created

**********/
```

```
void addNode(node**, polygon*);

#endif
```

2. decomputil.c

```
/*****************************************************************
FILE: decomposutil.c
PURPOSE: This file contains the functions which do the actual work
      for the decomposition.

*****************************************************************/



#include <stdlib.h>
#include <stdio.h>
#include "build.h"
#include "decomputil.h"
#include "util.h"

/*********************addEdge(event*, edge**)*****************************
This function adds edges to the active edge list

*************************************************************************/
int
addEdge(event* currentEvent, edge** activeEdges){

   edge* currentEdge;
   edge* previousEdge;
   int edgesAdded = 0;
   double delta = 1e-20;/*small number close to zero*/

   currentEdge = (*activeEdges);
   previousEdge= (*activeEdges);

   while((currentEdge) &&
       ((currentEdge->xPoint = intersect(currentEdge, currentEvent->eVertex))
        > (currentEvent->eVertex->posit.Y+delta))) {
      previousEdge = currentEdge;
      currentEdge = currentEdge->next;
   }

   /*identify convex and concave vertices, and simultaneous vertices now too*/
   if (order((currentEvent->eVertex->prev->posit),
          (currentEvent->eVertex->posit),
          (currentEvent->eVertex->next->posit)) > 0.0){
      currentEvent->eVertex->bndry[0]='V';
      if((currentEvent->eVertex->posit.X > currentEvent->eVertex->prev->posit.X)&&
        (currentEvent->eVertex->posit.X < currentEvent->eVertex->next->posit.X))
```

```
      currentEvent->eVertex->bndry[1]='O';
    else if ((previousEdge)&&
        ((previousEdge->leadingVertex->posit.X == currentEvent->eVertex->posit.X)||
        (previousEdge->trailingVertex->posit.X == currentEvent->eVertex->posit.X)))
      currentEvent->eVertex->bndry[1]='N';
    else
      currentEvent->eVertex->bndry[1]='O';
  }else{
    currentEvent->eVertex->bndry[0]='C';
    if(((currentEvent->eVertex->posit.X >= currentEvent->eVertex->prev->posit.X)&&
      (currentEvent->eVertex->posit.X <= currentEvent->eVertex->next->posit.X)) ||
      ((currentEvent->eVertex->posit.X > currentEvent->eVertex->prev->posit.X)&&
      (currentEvent->eVertex->posit.X > currentEvent->eVertex->next->posit.X)) ||
      ((currentEvent->eVertex->posit.X < currentEvent->eVertex->prev->posit.X)&&
      (currentEvent->eVertex->posit.X < currentEvent->eVertex->next->posit.X)))
      currentEvent->eVertex->bndry[1]='O';
    else if ((previousEdge)&&
        ((previousEdge->leadingVertex->posit.X == currentEvent->eVertex->posit.X)||
        (previousEdge->trailingVertex->posit.X == currentEvent->eVertex->posit.X)))
      currentEvent->eVertex->bndry[1]='N';
    else
      currentEvent->eVertex->bndry[1]='O';
  }

  /*maximal vertex; no edges added*/
  if ((currentEvent->eVertex->posit.X > currentEvent->eVertex->next->posit.X) &&
    (currentEvent->eVertex->posit.X > currentEvent->eVertex->prev->posit.X)){

      return edgesAdded;
    }

  /*examine edge defined by next vertex*/
  if (currentEvent->eVertex->posit.X < currentEvent->eVertex->next->posit.X){
    if (currentEdge == (*activeEdges)){
      currentEdge = (edge*)malloc(sizeof(edge));
      currentEdge->next = (*activeEdges);
      (*activeEdges) = currentEdge;
    }else{
      currentEdge=(edge*)malloc(sizeof(edge));
      currentEdge->next = previousEdge->next;
      previousEdge->next = currentEdge;
    }
    currentEdge->leadingVertex = currentEvent->eVertex;
    currentEdge->trailingVertex = currentEvent->eVertex->next;
    currentEdge->xPoint = currentEvent->eVertex->posit.Y;
    edgesAdded++;
  }

  /*examine edge defined by previouss vertex*/
  if (currentEvent->eVertex->posit.X < currentEvent->eVertex->prev->posit.X){
    if (edgesAdded && currentEvent->eVertex->prev->posit.Y <
```

```
          currentEvent->eVertex->next->posit.Y){
        previousEdge = currentEdge;
        currentEdge = currentEdge->next;
      }
      if (currentEdge == (*activeEdges)){
        currentEdge = (edge*)malloc(sizeof(edge));
        currentEdge->next = (*activeEdges);
        (*activeEdges) = currentEdge;
      }else{
        currentEdge=(edge*)malloc(sizeof(edge));
        currentEdge->next = previousEdge->next;
        previousEdge->next = currentEdge;
      }     currentEdge->leadingVertex = currentEvent->eVertex;
      currentEdge->trailingVertex = currentEvent->eVertex->prev;
      currentEdge->xPoint = currentEvent->eVertex->posit.Y;
      edgesAdded++;
    }
    return edgesAdded;

}

/**************************deleteEdge(event* edge**)**********************
This function deletes edges from the active edge list

*****************************************************************************/

int
deleteEdge(event* currentEvent, edge** activeEdges){

    edge* currentEdge;
    edge* previousEdge;
    int edgesDeleted = 0;

    currentEdge = (*activeEdges);
    previousEdge = (*activeEdges);

    while(currentEdge){

      if (currentEvent->eVertex == currentEdge->trailingVertex){

        if (currentEdge == (*activeEdges))
          (*activeEdges)=(*activeEdges)->next;
        else
          previousEdge->next = currentEdge->next;

        currentEdge->leadingVertex = NULL;
        currentEdge->trailingVertex = NULL;
        currentEdge->next = NULL;
        free(currentEdge);
        edgesDeleted++;
        if(*activeEdges)
```

```
            currentEdge = previousEdge->next;
          else
            currentEdge = NULL;
        }else{
          previousEdge=currentEdge;
          currentEdge=currentEdge->next;
        }
      }
      return edgesDeleted;
}

/**********************intersect(edge*, vertex*)*************************
This function computes the y-coordinate of the intersection of the
extension of the sweep line and the current edge. The x-coordinate is
the same as the current event.

*************************************************************************/
double
intersect(edge* currentEdge, vertex* currentEvertex){

   return(((((currentEdge->trailingVertex->posit.Y)-
         (currentEdge->leadingVertex->posit.Y))*
       ((currentEvertex->posit.X)-(currentEdge->leadingVertex->posit.X)))/
       ((currentEdge->trailingVertex->posit.X)-(currentEdge->leadingVertex->posit.X)))
       + currentEdge->leadingVertex->posit.Y);
}

/**********startCellUp(event*, polygon**, edge*, fence**, node*)********
This function finds the vertices for the start of an up cell

*************************************************************************/
void
startCellUp (event* cEvent, polygon** cellList,
         edge* activeEdges, fence** activeFences, node** cGraph){

   edge* cEdge;
   polygon* currentCell;
   vertex* curVertex;
   vertex* cellVertex;
   extern int cellCount;

   ++cellCount;

   /* put new cell on active list and make a node on the graph*/
   currentCell=(*cellList);
   (*cellList)=(polygon*)malloc(sizeof(polygon));
   (*cellList)->next = currentCell;
   currentCell=(*cellList);
   addNode(cGraph, currentCell);
   sprintf(currentCell->name,"C%d",cellCount);
   currentCell->mode = -1;
```

```
  /* add current event vertex to this new cell */
  curVertex = cEvent->eVertex;
  cellVertex = currentCell->vertexList =(vertex*)malloc(sizeof(vertex));
  cellVertex->posit.X = curVertex->posit.X;
  cellVertex->posit.Y = curVertex->posit.Y;
  sprintf(cellVertex->bndry,"\0");

  /* add another vertex */
  cellVertex->prev=(vertex*)malloc(sizeof(vertex));
  currentCell->open2 =curVertex->prev;/*bottom open vertex of this cell*/
  cellVertex->prev->next = cellVertex;
  cellVertex->prev->prev = NULL;
  cellVertex->prev->posit.X = curVertex->prev->posit.X;
  cellVertex->prev->posit.Y = curVertex->prev->posit.Y;
  sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);

  /*find appropriate edge which intersects the sweep line*/
  cEdge = activeEdges;
  while((cEdge->next->xPoint
        = intersect(cEdge->next,curVertex)) > curVertex->posit.Y)
    cEdge = cEdge->next;

  /* add this intersection point as a cell vertex */
  cellVertex->next=(vertex*)malloc(sizeof(vertex));
  cellVertex->next->prev = cellVertex;
  cellVertex->next->posit.X = curVertex->posit.X;
  cellVertex->next->posit.Y = cEdge->xPoint;
  sprintf(cellVertex->next->bndry,"\0");

  /* this edge will also be a cell adjacency edge */
  updateAdj(activeFences, currentCell,cellVertex->next,cellVertex,cGraph);

  /* add another vertex */
  cellVertex = cellVertex->next;
  cellVertex->next=(vertex*)malloc(sizeof(vertex));
  currentCell->open1 = cEdge->trailingVertex;/*top open vertex of this cell*/
  cellVertex->next->prev = cellVertex;
  cellVertex->next->next = NULL;
  cellVertex->next->posit.X = cEdge->trailingVertex->posit.X;
  cellVertex->next->posit.Y = cEdge->trailingVertex->posit.Y;
  sprintf(cellVertex->next->bndry,"%s",cEvent->owner->name);
}

/**********startCellDown(event*, polygon*, edge*, fence**, node**)*******
This function finds the vertices for the start of a down cell

**************************************************************************/
void
startCellDown (event* cEvent, polygon** cellList,
        edge* activeEdges, fence** activeFences, node** cGraph){
```

```
edge* cEdge;
polygon* currentCell;
vertex* curVertex;
vertex* nextVertex;
vertex* cellVertex;
extern int cellCount;

++cellCount;

/* put new cell on active list and make a node on the graph*/
currentCell=(*cellList);
(*cellList)=(polygon*)malloc(sizeof(polygon));
(*cellList)->next = currentCell;
currentCell=(*cellList);
addNode(cGraph, currentCell);
sprintf(currentCell->name,"C%d", cellCount);
currentCell->mode = -1;

/* add current event vertex to this cell */
curVertex = nextVertex = cEvent->eVertex;
cellVertex = currentCell->vertexList=(vertex*)malloc(sizeof(vertex));
cellVertex->posit.X = curVertex->posit.X;
cellVertex->posit.Y = curVertex->posit.Y;
sprintf(cellVertex->bndry,"\0");

/* add another vertex */
cellVertex->next=(vertex*)malloc(sizeof(vertex));
currentCell->open1 = curVertex->next;/*top open vertex for this cell*/
cellVertex->next->prev = cellVertex;
cellVertex->next->next = NULL;
cellVertex->next->posit.X = curVertex->next->posit.X;
cellVertex->next->posit.Y = curVertex->next->posit.Y;
sprintf(cellVertex->next->bndry,"%s",cEvent->owner->name);

do{/* do this for all simultaneous vertices below the current one */
   curVertex = nextVertex;
   cEvent = cEvent->next;
   nextVertex = cEvent->eVertex;

   /* find appropriate edge which intersects the sweep line */
   cEdge = activeEdges;
   while((cEdge->xPoint = intersect(cEdge,curVertex)) >= curVertex->posit.Y)
      cEdge = cEdge->next;

   /* add this intersection point as a cell vertex */
   cellVertex->prev=(vertex*)malloc(sizeof(vertex));
   cellVertex->prev->next = cellVertex;
   cellVertex->prev->posit.X = curVertex->posit.X;
   cellVertex->prev->posit.Y = cEdge->xPoint;
```

```
      /* will edge be an adjacency edge? */
      if((curVertex->prev == nextVertex)&&
         (cellVertex->prev->posit.X == nextVertex->posit.X))
         sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);
      else{
         sprintf(cellVertex->prev->bndry,"\0");
         updateAdj(activeFences, currentCell,
            cellVertex,cellVertex->prev,cGraph);
      }
      cellVertex = cellVertex->prev;
   }while (cEdge->trailingVertex->posit.X == curVertex->posit.X);

   /* add another vertex */
   currentCell->open2 = cEdge->trailingVertex;/*bottom open vertex of cell*/
   cellVertex->prev=(vertex*)malloc(sizeof(vertex));
   cellVertex->prev->next = cellVertex;
   cellVertex->prev->prev = NULL;
   cellVertex->prev->posit.X = cEdge->trailingVertex->posit.X;
   cellVertex->prev->posit.Y = cEdge->trailingVertex->posit.Y;
   sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);
}

/*********startCellUpDown(event*, polygon*, edge*, fence*, node**)*********
This function finds the vertices for the start of an up/down cell

*****************************************************************************/
void
startCellUpDown (event* cEvent, polygon** cellList,
          edge* activeEdges, fence** activeFences, node** cGraph){

   edge* cEdge;
   polygon* currentCell;
   vertex* curVertex;
   vertex* nextVertex;
   vertex* cellVertex;
   extern int cellCount;

   ++cellCount;

   /*put new cell on active list and make a node on the graph*/
   currentCell=(*cellList);
   (*cellList)=(polygon*)malloc(sizeof(polygon));
   (*cellList)->next = currentCell;
   currentCell=(*cellList);
   addNode(cGraph, currentCell);
   currentCell->mode = -1;
   sprintf(currentCell->name,"C%d", cellCount);

   /* add current event vertex to this cell */
   curVertex = cEvent->eVertex;
   cellVertex = currentCell->vertexList = (vertex*)malloc(sizeof(vertex));
```

```
cellVertex->posit.X = curVertex->posit.X;
cellVertex->posit.Y = curVertex->posit.Y;
sprintf(cellVertex->bndry,"\0");

/* find appropriate edge which intersects the sweep line */
cEdge = activeEdges;   while((cEdge->next->xPoint
    =intersect(cEdge->next,curVertex)) > curVertex->posit.Y)
  cEdge = cEdge->next;

/* add this intersection point as a cell vertex */
cellVertex->next=(vertex*)malloc(sizeof(vertex));
cellVertex->next->prev = cellVertex;
cellVertex->next->posit.X = curVertex->posit.X;
cellVertex->next->posit.Y = cEdge->xPoint;
sprintf(cellVertex->next->bndry,"\0");

/* this edge will also be a cell adjacency edge */
updateAdj(activeFences, currentCell,cellVertex->next,cellVertex,cGraph);

/* add another vertex */
cellVertex = cellVertex->next;
cellVertex->next=(vertex*)malloc(sizeof(vertex));
currentCell->open1 = cEdge->trailingVertex;/*top open vertex for cell*/
cellVertex->next->prev = cellVertex;
cellVertex->next->next = NULL;
cellVertex->next->posit.X = cEdge->trailingVertex->posit.X;
cellVertex->next->posit.Y = cEdge->trailingVertex->posit.Y;
sprintf(cellVertex->next->bndry,"%s",cEvent->owner->name);

curVertex= nextVertex = cEvent->eVertex;
cellVertex = currentCell->vertexList;
do{/* do this for all simultaneous vertices */
  curVertex = nextVertex;
  cEvent = cEvent->next;
  nextVertex = cEvent->eVertex;

  /* find appropriate edge which intersects the sweep line */
  cEdge = activeEdges;
  while((cEdge->xPoint = intersect(cEdge,curVertex)) >= curVertex->posit.Y)
    cEdge = cEdge->next;

  /* add this intersection point as a cell vertex */
  cellVertex->prev=(vertex*)malloc(sizeof(vertex));
  cellVertex->prev->next = cellVertex;
  cellVertex->prev->posit.X = curVertex->posit.X;
  cellVertex->prev->posit.Y = cEdge->xPoint;

  /* will edge be an adjacency edge too? */
  if(curVertex->prev == nextVertex)
    sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);
  else{
```

```
        sprintf(cellVertex->prev->bndry,"\0");
        updateAdj(activeFences, currentCell,
          cellVertex,cellVertex->prev,cGraph);
      }
      cellVertex = cellVertex->prev;
    }while (cEdge->trailingVertex->posit.X == curVertex->posit.X);

    /*add another vertex */
    cellVertex->prev=(vertex*)malloc(sizeof(vertex));
    currentCell->open2 = cEdge->trailingVertex;/*bottom open vertex of cell*/
    cellVertex->prev->next = cellVertex;
    cellVertex->prev->prev = NULL;
    cellVertex->prev->posit.X = cEdge->trailingVertex->posit.X;
    cellVertex->prev->posit.Y = cEdge->trailingVertex->posit.Y;
    sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);

}

/************startCellIsolated(event*, polygon**, node**)**************
ADD ISOLATED CELL TO WORKING LIST
**********************************************************************/
void
startCellIsolated (event* cEvent, polygon** cellList, node** cGraph){

    polygon* currentCell;
    edge* cEdge;
    vertex* curVertex;
    vertex* nextVertex;
    vertex* cellVertex;
    extern int cellCount;

    ++cellCount;

    /* add new cell to active list */
    currentCell=(*cellList);
    (*cellList)=(polygon*)malloc(sizeof(polygon));
    (*cellList)->next = currentCell;
    currentCell=(*cellList);
    addNode(cGraph, currentCell);
    currentCell->mode = -1;
    sprintf(currentCell->name,"C%d", cellCount);

    /* add current event vertex to this cell */
    curVertex = cEvent->eVertex;
    cellVertex = currentCell->vertexList = (vertex*)malloc(sizeof(vertex));
    cellVertex->posit.X = curVertex->posit.X;
    cellVertex->posit.Y = curVertex->posit.Y;
    sprintf(cellVertex->bndry,"%s",cEvent->owner->name);

    /* add another vertex */
    cellVertex->next=(vertex*)malloc(sizeof(vertex));
```

```
  currentCell->open1 = curVertex->next;/*top open vertex of cell*/
  cellVertex->next->prev = cellVertex;
  cellVertex->next->next = NULL;
  cellVertex->next->posit.X = curVertex->next->posit.X;
  cellVertex->next->posit.Y = curVertex->next->posit.Y;
  sprintf(cellVertex->next->bndry,"\0");

  /* add another vertex */
  cellVertex->prev=(vertex*)malloc(sizeof(vertex));
  currentCell->open2 = curVertex->prev;/*bottom open vertex of cell*/
  cellVertex->prev->next = cellVertex;
  cellVertex->prev->prev = NULL;
  cellVertex->prev->posit.X = curVertex->prev->posit.X;
  cellVertex->prev->posit.Y = curVertex->prev->posit.Y;
  sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);

}

/******finishCellUp(event*, polygon**, edge*, world**, fence**)***************
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

*****************************************************************************/
void
finishCellUp (event* cEvent, polygon** cellList,
        edge* activeEdges, world** decompWorld, fence** activeFences){

  edge* cEdge;
  polygon* currentCell;
  polygon* prevCell;
  vertex* curVertex;
  vertex* cellPendVertex;
  vertex* cellNendVertex;
  fence* newFence;

  prevCell = currentCell = (*cellList);
  curVertex = cEvent->eVertex;
  cEdge = activeEdges;
  while((cEdge->next->xPoint
     =intersect(cEdge->next,curVertex)) > curVertex->posit.Y)
    cEdge = cEdge->next;

  /* find cell on active list which is to be completed by this vertex*/
  while (currentCell->open1 != cEdge->trailingVertex){
    prevCell = currentCell;
    currentCell = currentCell->next;
  }

  /* move a pointer to both open vertices */
  cellPendVertex = cellNendVertex = currentCell->vertexList;
  while (cellPendVertex->prev)
```

```
    cellPendVertex = cellPendVertex->prev;

  while (cellNendVertex->next)
    cellNendVertex = cellNendVertex->next;

  /* link up open vertices */
  cellNendVertex->next = cellPendVertex;
  cellNendVertex->next->prev = cellNendVertex;
  cellNendVertex->posit.Y = cEdge->xPoint;
  cellNendVertex->posit.X = curVertex->posit.X;
  sprintf(cellNendVertex->bndry,"\0");

  /* this edge is an adjacency edge */
  putFence(activeFences, currentCell, cellNendVertex, cellPendVertex);

  /* add completed cell to the world */
  if (prevCell != currentCell)
    prevCell->next = currentCell->next;
  else
    (*cellList) = currentCell->next;

  currentCell->next = (*decompWorld)->polygonList;
  (*decompWorld)->polygonList = currentCell;

}

/******finishCellDown(event*, polygon**, edge*, world**, fence**, int)******
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

**********************************************************************/
void
finishCellDown (event* cEvent, polygon** cellList,
        edge* activeEdges, world** decompWorld, fence** activeFences,
        int concave){

  edge* cEdge;
  polygon* currentCell;
  polygon* prevCell;
  vertex* curVertex;
  vertex* nextVertex;
  vertex* cellPendVertex;
  vertex* cellNendVertex;

  prevCell = currentCell = (*cellList);
  curVertex = nextVertex = cEvent->eVertex;

  /* find cell on active list which isto be completed by this vertex*/
  while (currentCell->open1 != curVertex){
    prevCell = currentCell;
    currentCell = currentCell ->next;
```

```
}

/* move a pointer to both open vertices*/
cellPendVertex = cellNendVertex = currentCell->vertexList;
while (cellNendVertex->next)
   cellNendVertex = cellNendVertex->next;

while (cellPendVertex->prev)
   cellPendVertex = cellPendVertex->prev;

do{/* do this for all simultaneous events*/
   curVertex = nextVertex;
   cEvent = cEvent->next;
   nextVertex = cEvent->eVertex;

   /* add simultaneous event vertices to this cell */
   cEdge = activeEdges;
   while((cEdge->xPoint = intersect(cEdge,curVertex)) >= curVertex->posit.Y)
      cEdge = cEdge->next;
   cellNendVertex->next=(vertex*)malloc(sizeof(vertex));
   cellNendVertex->next->prev = cellNendVertex;
   cellNendVertex->next->posit.X = curVertex->posit.X;
   cellNendVertex->next->posit.Y = cEdge->xPoint;

   /* is this added edge an adjacency edge too? */
   if((curVertex->next == nextVertex)&&
      (cellNendVertex->next->posit.X == nextVertex->posit.X))
       sprintf(cellNendVertex->bndry,"%s",cEvent->owner->name);
   else{
       sprintf(cellNendVertex->next->bndry,"\0");
       putFence(activeFences,currentCell,
            cellNendVertex,cellNendVertex->next);
   }
   cellNendVertex = cellNendVertex->next;
}while ((concave && ((curVertex->posit.X == curVertex->next->posit.X)&&
      (curVertex->next->posit.X < curVertex->next->next->posit.X)))||
      ((cEdge->leadingVertex->posit.X == curVertex->posit.X) &&
      ((nextVertex->posit.X <= nextVertex->prev->posit.X)&&
      (nextVertex->posit.X <= nextVertex->next->posit.X))));

/*link up open vertices */
cellNendVertex->next = cellPendVertex->next;
cellPendVertex->next->prev = cellNendVertex;

/* add completed cell to the world */
if (prevCell != currentCell)
   prevCell->next = currentCell->next;
else
   (*cellList) = currentCell->next;

currentCell->next = (*decompWorld)->polygonList;
```

```
   (*decompWorld)->polygonList = currentCell;

}

/*****finishCellUpDown(event*, polygon**, edge*, world**, fence*)*******
This function locates which cell on the active list is completed by this
event, adds the apprpriate vertices, and adds new cell to the world

**************************************************************************/
void
finishCellUpDown (event* cEvent, polygon** cellList,
        edge* activeEdges, world** decompWorld, fence** activeFences){

   edge* topEdge;
   edge* bottomEdge;
   polygon* currentCell;
   polygon* prevCell;
   vertex* curVertex;
   vertex* nextVertex;
   vertex* cellPendVertex;
   vertex* cellNendVertex;

   prevCell = currentCell = (*cellList);
   curVertex = nextVertex = cEvent->eVertex;
   topEdge = bottomEdge = activeEdges;

   while((topEdge->next->xPoint
       =intersect(topEdge->next,curVertex)) > curVertex->posit.Y)
     topEdge = topEdge->next;

   /* find cel on active list which is to be completed by this vertex*/
   while (currentCell->open1 != topEdge->trailingVertex){
     prevCell = currentCell;
     currentCell = currentCell ->next;
   }

   /* move a pointer to both open vertices */
   cellPendVertex = cellNendVertex = currentCell->vertexList;
   while (cellNendVertex->next)
     cellNendVertex = cellNendVertex->next;

   while (cellPendVertex->prev)
     cellPendVertex = cellPendVertex->prev;

   /* add another vertex */
   cellNendVertex->next = malloc(sizeof(vertex));
   cellNendVertex->next->prev = cellNendVertex;
   cellNendVertex->posit.X = curVertex->posit.X;
   cellNendVertex->posit.Y = topEdge->xPoint;
   sprintf(cellNendVertex->bndry,"\0");
```

```
    /* update the top open vertex */
    cellNendVertex->next->posit.X = curVertex->posit.X;
    cellNendVertex->next->posit.Y = curVertex->posit.Y;
    sprintf(cellNendVertex->next->bndry,"\0");

    /* this edge is an adjacency edge */
    putFence(activeFences, currentCell, cellNendVertex, cellNendVertex->next);

    do{/* do for all simultaneous vertices */
       cellNendVertex = cellNendVertex->next;
       curVertex = nextVertex;
       cEvent = cEvent->next;
       nextVertex = cEvent->eVertex;
       bottomEdge = activeEdges;
       while((bottomEdge->xPoint =
            intersect(bottomEdge,curVertex)) >= curVertex->posit.Y)
          bottomEdge = bottomEdge->next;
       cellNendVertex->next=(vertex*)malloc(sizeof(vertex));
       cellNendVertex->next->prev = cellNendVertex;
       cellNendVertex->next->posit.X = curVertex->posit.X;
       cellNendVertex->next->posit.Y = bottomEdge->xPoint;

       /* is this edge an adjacency edge too? */
       if((curVertex->next == nextVertex)&&
          (cellNendVertex->posit.X == nextVertex->posit.X))
          sprintf(cellNendVertex->next->bndry,"%s",cEvent->owner->name);
       else{
          sprintf(cellNendVertex->next->bndry,"\0");
          putFence(activeFences,currentCell,
             cellNendVertex,cellNendVertex->next);
       }
    }while ((bottomEdge->leadingVertex->posit.X == curVertex->posit.X) &&
          ((nextVertex->posit.X <= nextVertex->prev->posit.X)&&
          (nextVertex->posit.X <= nextVertex->next->posit.X)));

    /* link up open vertices */
    cellNendVertex->next->next = cellPendVertex->next;
    cellPendVertex->next->prev = cellNendVertex->next;

    /* add completed cell to the world */
    if (prevCell != currentCell)
       prevCell->next = currentCell->next;
    else
       (*cellList) = currentCell->next;
    currentCell->next = (*decompWorld)->polygonList;
    (*decompWorld)->polygonList = currentCell;

}

/*****finishCellIsolated(event*, polygon**, world**)*****************
This function locates which cell on the active list is completed by this
```

```
event, adds the apprpriate vertices, and adds new cell to the world

**********/
void
finishCellIsolated (event* cEvent, polygon** cellList,world** decompWorld){

   polygon* currentCell;
   polygon* prevCell;
   vertex* curVertex;
   vertex* cellPendVertex;
   vertex* cellNendVertex;

   prevCell = currentCell = (*cellList);
   curVertex = cEvent->eVertex;

   /* find cell on active list to be completed by this event */
   while((currentCell->open1 != curVertex)
         || (currentCell->open2 != curVertex)){
      prevCell = currentCell;
      currentCell = currentCell->next;
    }

   /* move a pointer to both open vertices */
   cellPendVertex = cellNendVertex = currentCell->vertexList;
   while (cellPendVertex->prev)
      cellPendVertex = cellPendVertex->prev;

   while (cellNendVertex->next)
      cellNendVertex = cellNendVertex->next;

   /* link up open vertices */
   cellNendVertex->next = cellPendVertex->next;
   cellPendVertex->next->prev = cellNendVertex;
   sprintf(cellPendVertex->bndry,"%s",cEvent->owner->name);
   sprintf(cellPendVertex->next->bndry,"%s",cEvent->owner->name);
   cellPendVertex->next = cellPendVertex->prev = NULL;
   free(cellPendVertex);

   /* add completed cell to the world */
   if (prevCell != currentCell)
      prevCell->next = currentCell->next;
   else
      (*cellList) = currentCell->next;

   currentCell->next = (*decompWorld)->polygonList;
   (*decompWorld)->polygonList = currentCell;

}

/**********extendCell(event* polygon**)**********
This function locates which cell on the active list needs to be
```

```
extended by this concave vertex

**********************************************************************/
void
extendCell(event* cEvent, polygon** cellList){

   polygon* currentCell;
   vertex* cellVertex;
   vertex* curVertex;

   currentCell = (*cellList);
   curVertex = cEvent->eVertex;

   /* find the correct cell */
   while ((currentCell->open1 != curVertex)
         && (currentCell->open2 != curVertex))
      currentCell = currentCell->next;
   cellVertex = currentCell->vertexList;

   /* is this the top of a cell ?*/
   if ((curVertex->posit.X >= curVertex->next->posit.X)
     &&(curVertex->posit.X < curVertex->prev->posit.X)){
      while(cellVertex->prev)
         cellVertex = cellVertex->prev;
      cellVertex->prev = malloc(sizeof(vertex));
      cellVertex->prev->prev = NULL;
      cellVertex->prev->next = cellVertex;
      cellVertex->prev->posit.X = curVertex->prev->posit.X;
      cellVertex->prev->posit.Y = curVertex->prev->posit.Y;
      currentCell->open2 = curVertex->prev;
      sprintf(cellVertex->bndry,"%s",cEvent->owner->name);
      sprintf(cellVertex->prev->bndry,"%s",cEvent->owner->name);
   }else{/* or the botom? */
      while(cellVertex->next)
         cellVertex = cellVertex->next;
      cellVertex->next = malloc(sizeof(vertex));
      cellVertex->next->next = NULL;
      cellVertex->next->prev = cellVertex;
      cellVertex->next->posit.X = curVertex->next->posit.X;
      cellVertex->next->posit.Y = curVertex->next->posit.Y;
      currentCell->open1 = curVertex->next;
      sprintf(cellVertex->bndry,"%s",cEvent->owner->name);
      sprintf(cellVertex->next->bndry,"%s",cEvent->owner->name);   }
}

/******putFence(fence**, polygon*, vertex*, vertex*)*******************
This function puts an adjacency fence on a list to be matched later

**********************************************************************/
void
putFence(fence** fenceList, polygon* cell,
```

```
        vertex* topVertex, vertex* bottomVertex){

    fence* newFence;

    newFence = (fence*)malloc(sizeof(fence));
    newFence->owner = cell;
    newFence->topVertex = topVertex;
    newFence->bottomVertex = bottomVertex;

    newFence->next = (*fenceList);
    (*fenceList) = newFence;

}

/**********updateAdj(fence**, polygon*, vertex*, vertex*, node**)*********
This function matches an adjacency fence with one already on the list
to determine cell adjacency, and then updates the adjacency graph

**************************************************************************/
void
updateAdj(fence** fenceList, polygon* curCell,
          vertex* topVertex, vertex* bottomVertex, node** cGraph){

    fence* curFence;
    fence* prevFence;
    node* firstNode;
    node* secondNode;
    arc* cArc;


    /* find the matching fence */
    curFence = prevFence = (*fenceList);
    while((curFence->topVertex->posit.X != topVertex->posit.X)||
        (curFence->topVertex->posit.Y != topVertex->posit.Y)||
        (curFence->bottomVertex->posit.X != bottomVertex->posit.X)||
        (curFence->bottomVertex->posit.Y != bottomVertex->posit.Y)){
      prevFence = curFence;
      curFence = curFence ->next;
    }

    /* deleted the matched fence from the list */
    if(curFence == (*fenceList))
      (*fenceList)=(*fenceList)->next;
    else
      prevFence->next = curFence->next;

    /* find the appropriate nodes of the graph */
    firstNode = (*cGraph);
    while(firstNode->cell != curCell)
      firstNode = firstNode->next;
```

```
    secondNode = (*cGraph);
    while(secondNode->cell != curFence->owner)
      secondNode = secondNode->next;

    /* add arcs to both nodes on the graph */
    cArc = (arc*)malloc(sizeof(arc));
    cArc->next = firstNode->arcList;
    firstNode->arcList = cArc;
    cArc->Node = secondNode;
    cArc->visited = 0;
    sprintf(bottomVertex->bndry,"%s",curFence->owner->name);

    cArc = (arc*)malloc(sizeof(arc));
    cArc->next = secondNode->arcList;
    secondNode->arcList = cArc;
    cArc->Node = firstNode;
    cArc->visited = 0;
    sprintf(curFence->topVertex->bndry,"%s",curCell->name);
}

/**********addNode(node**, polygon**)**********
This function adds a node to the connectivity graph whenever a cell
is created

**********/
void
addNode(node** cGraph, polygon* curCell){

    node* newNode;

    newNode = (*cGraph);
    (*cGraph) = (node*)malloc(sizeof(node));
    (*cGraph)->next = newNode;
    newNode = (*cGraph);
    newNode->cell = curCell;
    newNode->arcList = NULL;
    newNode->predecessor = NULL;
    newNode->curArc = NULL;

}
```

E. MISCELANEOUS UTILITIES

1. util.h

```
/**********
FILE: util.h
PURPOSE: This file contains some small utility functions used
      by other functions in this program
**********/
```

```
#ifndef __world_util_h
#define __world_util_h

#include "build.h"

/*********************************************************************
PURPOSE: returns 0.0 if the three points are colinear, < 0.0 if
      they are clockwise, and > 0.0 if they are counterclockwise
*********************************************************************/
double order(point, point, point);

/*********************************************************************
PURPOSE: frees memory allocated during world decompostion
*********************************************************************/
void freeWorld(world**, world**, event**, node**);

/*********************************************************************
PURPOSE:  reads either the world name or poygon name from the inputfile
*********************************************************************/
void getName(FILE*, char*, int);

#endif
```

2. util.c

```
/*********************************************************************
FILE: util.c
PURPOSE: This file contains some small utility functions used
      by other functions in this program
*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "util.h"

/**********************getName(FILE*, char*)*****************************
This function reads the characters from the input file which correspond
to a world name, or a polygon name
*********************************************************************/
void
getName(FILE* input_file, char* name, int length)
{
int counter1;
int counter2;
char letter;
char tempName[100];

counter1 =0;
```

```
 while(1){
  if ((letter = getc(input_file)) == '(')
   break;
  else if((letter > '!') && (letter < '~')){
   tempName[counter1] = letter;
   counter1++;
  }
 }
 for (counter2=0;((counter2 < (length-1))&&(counter2 != counter1));counter2++)
  name[counter2]=tempName[counter2];
 name[counter2]= '\0';
}
/************************order(point, point, point)**********************
This function computes whether the orientation of the points is colinear,
clockwise, or counter-clockwise. If the points are colinear, the function
returns 0. If the points are clockwise, the function returns a number < 0.
If the points are counter-clockwise, the function returns a number > 0
**************************************************************************/
double
order (point first, point second, point third)

{
 return ((((second.X - first.X)*(third.Y - first.Y))
      -((third.X - first.X)*(second.Y - first.Y)))/2.0);
}

/********************freeWorld(world**, world**, event**, node**)*********
Thie function frees the allocated memory for one decomposition before
another decomposition is attempted
**************************************************************************/
void
freeWorld(world** origWorld, world** decWorld,
          event** eventList, node** cGraph)
{

   polygon* curPolygon;
   polygon* nextPolygon;
   vertex* curVertex;
   vertex* nextVertex;
   node* curNode;
   node* nextNode;
   arc* curArc;

   curPolygon = (*origWorld)->polygonList;

      do{
         curVertex = curPolygon->vertexList;
         curVertex->prev->next = NULL;
         do{
                  nextVertex =curVertex->next;
```

```
                free(curVertex);
                curVertex = nextVertex;

        }while (curVertex);

        nextPolygon = curPolygon->next;
        free(curPolygon);
        curPolygon = nextPolygon;

    }while(curPolygon != (*origWorld)->polygonList);

 if (*decWorld){
    curPolygon = (*decWorld)->polygonList;

        do{
        curVertex = curPolygon->vertexList;
        curVertex->prev->next = NULL;
        do{
           nextVertex =curVertex->next;
           free(curVertex);
           curVertex = nextVertex;

        }while (curVertex);

        nextPolygon = curPolygon->next;
        free(curPolygon);
        curPolygon = nextPolygon;

    }while(curPolygon != (*decWorld)->polygonList);

 }

 if (*cGraph){
    curNode = (*cGraph);

    do{
       curArc = curNode->arcList;
       do{
          curNode->arcList = curArc->next;
          free(curArc);
          curArc = curNode->arcList;
       }while(curArc);
       nextNode = curNode->next;
       free(curNode);
       curNode = nextNode;
    }while(curNode);
 }

 free((*origWorld));
 free((*decWorld));
 free((*eventList));
```

```
    free((*cGraph));
    (*eventList) = NULL;
    (*cGraph) = NULL;
}
```

F. CREATE WORLD FILES

1. worldfile.h

```
/*************************************************************************
FILE: worldfile.h
PURPOSE: This file contains a function prototype
*************************************************************************/


#ifndef __worldfile_h
#define __worldfile_h

#include "build.h"

/********createWorldFile(world*, node*, char*, double*, int*,char*)*******
This function makes the file which defines the cells and vertices of
the world, and the nodes and arcs of the graph. It also makes two
functions which intialize these values
*************************************************************************/

void createWorldFile(world*, node*, char*, double, int, char*);

#endif
```

2. worldfile.c

```
/*************************************************************************
FILE: worldfile.c
PURPOSE: This file contains the function which creates the C files
      that are used to link the world representation into yamabico's
      kernel.

*************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "build.h"
```

```
/********createWorldFile(world*, node*, char*, double*, int*,char*)*******
This function makes the file which defines the cells and vertices of
the world, and the nodes and arcs of the graph. It also makes two
functions which intialize these values
*****************************************************************************/
void
createWorldFile(world* robotWorld, node* cGraph, char* fileName,
        double sweepAngle,int worldNumber, char* worldTag){

  FILE* worldCfile;

  polygon* curPolygon;
  vertex* curVertex;
  node* curNode;
  node* adjNode;
  arc* curArc;

  double cosRotInv;
  double sinRotInv;
  double rotX, rotY;

  int counter1 =1;
  int counter2 =1;
  int counter3 =1;

  cosRotInv = cos((-(M_PI*(90.0-sweepAngle)))/180.0);
  sinRotInv = sin((-(M_PI*(90.0-sweepAngle)))/180.0);
  worldCfile = fopen(fileName,"w");

  fprintf(worldCfile,"/* This file is generated from world information\n");
  fprintf(worldCfile," and must be compiled and linked into Yamabico's\n");
  fprintf(worldCfile," kernel. The file \"worlds.h\" contains the \n");
  fprintf(worldCfile," declarations which allow access to the \n");
  fprintf(worldCfile," world model */\n\n");
  fprintf(worldCfile,"#include <stdlib.h>\n");
  fprintf(worldCfile,"#include <stdio.h>\n");
  fprintf(worldCfile,"#include <string.h>\n");
  fprintf(worldCfile,"#include \"build.h\"\n\n");
  fprintf(worldCfile,"/*declaring structure for world*/\n\n");
  fprintf(worldCfile,"world robotsWorld_%s;\n\n",worldTag);
  fprintf(worldCfile,"/*delcaring structures for polygons and vertices*/\n");
  counter1 = 1;
  curPolygon = robotWorld->polygonList;
  do{
    fprintf(worldCfile,"\npolygon  poly%d_%d;\n",worldNumber,counter1);
    counter2 = 1;
    curVertex = curPolygon->vertexList;
    do{
      fprintf(worldCfile,"vertex  vert%d_%d_%d;\n",worldNumber,
          counter1, counter2);
      curVertex = curVertex->next;
```

```
      ++counter2;
    }while(curVertex != curPolygon->vertexList);
    curPolygon = curPolygon->next;
    ++counter1;
  }while(curPolygon != robotWorld->polygonList);

  /* variables for connectivity graph */
  if (cGraph){
    fprintf(worldCfile,"\n\n/*declaring first node for graph*/\n\n");
    fprintf(worldCfile,"node* worldGraph_%s;\n\n",worldTag);
    fprintf(worldCfile,"/*delcaring structures for nodes and arcs*/\n");
    counter1 = 1;
    curNode = cGraph;
    do{
      fprintf(worldCfile,"\nnode  node%d_%d;\n",worldNumber,counter1);
      counter2 = 1;
      curArc = curNode->arcList;
      do{
        fprintf(worldCfile,"arc   arc%d_%d_%d;\n",worldNumber,
          counter1, counter2);
        curArc = curArc->next;
        ++counter2;
      }while(curArc);
      curNode = curNode->next;
      ++counter1;
    }while(curNode);
  }

  /*initialization function for world representation*/
  fprintf(worldCfile,"\n/*assigning values for polygons and vertices*/\n");

  fprintf(worldCfile,"\n\nInitializeWorld_%s(){\n",worldTag);
  fprintf(worldCfile,"\n robotsWorld_%s.polygonList = &poly%d_1; \n",
    worldTag, worldNumber);
  counter1 = 1;
  curPolygon = robotWorld->polygonList;
  do{
    fprintf(worldCfile,"\n  strcpy(poly%d_%d.name, \"%s\");\n",
      worldNumber, counter1,curPolygon->name);
    fprintf(worldCfile,"  poly%d_%d.mode = %d;\n",worldNumber,
      counter1,curPolygon->mode);
    fprintf(worldCfile,"  poly%d_%d.vertexList = &vert%d_%d_1;\n",
      worldNumber,counter1, worldNumber,counter1);
    if(curPolygon->next != robotWorld->polygonList)
      fprintf(worldCfile,"  poly%d_%d.next = &poly%d_%d;\n",
        worldNumber,counter1,worldNumber,(counter1+1));
    else
      fprintf(worldCfile,"  poly%d_%d.next = &poly%d_1;\n",
        worldNumber,counter1,worldNumber);

    counter2 = 1;
```

```
  curVertex = curPolygon->vertexList;
  do{
    rotX = ((curVertex->posit.X*cosRotInv)-(curVertex->posit.Y*sinRotInv));
    rotY = ((curVertex->posit.X*sinRotInv)+(curVertex->posit.Y*cosRotInv));
    fprintf(worldCfile,"    vert%d_%d_%d.posit.X = %6.5f;\n",
      worldNumber,counter1,counter2,rotX);
    fprintf(worldCfile,"    vert%d_%d_%d.posit.Y = %6.5f;\n",
      worldNumber,counter1,counter2,rotY);
    fprintf(worldCfile,"    strcpy(vert%d_%d_%d.bndry,\"%s\");\n",
      worldNumber,counter1,counter2,curVertex->bndry);
    if(curVertex->next != curPolygon->vertexList)
      fprintf(worldCfile,"    vert%d_%d_%d.next = &vert%d_%d_%d;\n"
        ,worldNumber,counter1,counter2,
        worldNumber,counter1,(counter2+1));
    else
      fprintf(worldCfile,"    vert%d_%d_%d.next = &vert%d_%d_1;\n"
        ,worldNumber, counter1,counter2,
        worldNumber,counter1);

    if(curVertex != curPolygon->vertexList)
      fprintf(worldCfile,"    vert%d_%d_%d.prev = &vert%d_%d_%d;\n"
        ,worldNumber,counter1,counter2,
        worldNumber,counter1,(counter2-1));
    curVertex = curVertex->next;
    ++counter2;
  }while(curVertex != curPolygon->vertexList);
  fprintf(worldCfile,"    vert%d_%d_1.prev = &vert%d_%d_%d;\n",
    worldNumber,counter1,worldNumber,counter1,(counter2-1));
  curPolygon = curPolygon->next;
  ++counter1;
}while(curPolygon != robotWorld->polygonList);

fprintf(worldCfile,"}\n");

/*initialization function for connectivity graph */
if (cGraph){
  fprintf(worldCfile,"\n/*assigning values for node and arcs*/\n");

  fprintf(worldCfile,"\n\nInitializeGraph_%s(){\n",worldTag);
  fprintf(worldCfile,"\n worldGraph_%s = &node%d_1; \n",
            worldTag, worldNumber);      counter1 = 1;
  curNode = cGraph;
  do{
    counter3=1;
    curPolygon = robotWorld->polygonList;
    while(curPolygon != curNode->cell){
      counter3++;
      curPolygon = curPolygon->next;
    }
    fprintf(worldCfile,"  node%d_%d.cell = &poly%d_%d;\n",
      worldNumber,counter1, worldNumber,counter3);
```

```
        fprintf(worldCfile,"  node%d_%d.arcList = &arc%d_%d_1;\n",
          worldNumber,counter1,worldNumber,counter1);
        fprintf(worldCfile,"  node%d_%d.predecessor = NULL;\n", worldNumber,counter1);
        fprintf(worldCfile,"  node%d_%d.curArc = NULL;\n", worldNumber,counter1);
        if(curNode->next)
          fprintf(worldCfile,"  node%d_%d.next = &node%d_%d;\n",
            worldNumber,counter1,worldNumber,(counter1+1));
        else
          fprintf(worldCfile,"   node%d_%d.next = NULL;\n",
            worldNumber,counter1);
        counter2 = 1;          curArc = curNode->arcList;
        do{
          counter3 = 1;
          /*which polygon number corresesponds to this cell?*/
          adjNode = cGraph;
          while(curArc->Node != adjNode){
            counter3++;
            adjNode = adjNode->next;
          }
          fprintf(worldCfile,"     arc%d_%d_%d.Node = &node%d_%d;\n",
            worldNumber,counter1,counter2, worldNumber,counter3);
          fprintf(worldCfile,"     arc%d_%d_%d.visited = 0;\n",
            worldNumber,counter1,counter2);
          if(curArc->next)
            fprintf(worldCfile,"     arc%d_%d_%d.next = &arc%d_%d_%d;\n",
              worldNumber,counter1,counter2,
              worldNumber,counter1,(counter2+1));
          else
            fprintf(worldCfile,"     arc%d_%d_%d.next = NULL;\n",
              worldNumber,counter1,counter2);
          curArc = curArc->next;
          ++counter2;
        }while(curArc);
        curNode = curNode->next;
        ++counter1;
      }while(curNode);

      fprintf(worldCfile,"}\n");
    }
    fclose(worldCfile);
    printf("The world for the %3.2f degree sweep is in the file: %s\n",
          sweepAngle, fileName);
    printf("and has the varialble name of: robotsWorld_%s\n",worldTag);
}
```

3. sample world file

```
/* This file is generated from world information
 and must be compiled and linked into Yamabico's
 kernel. The file "worlds.h" contains the
```

```
 declarations which allow access to the
 world model */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "build.h"

/*declaring structure for world*/

world robotsWorld_180_00;

/*delcaring structures for polygons and vertices*/

polygon  poly2_1;
vertex  vert2_1_1;
vertex  vert2_1_2;
vertex  vert2_1_3;
vertex  vert2_1_4;
vertex  vert2_1_5;
vertex  vert2_1_6;

polygon  poly2_2;
vertex  vert2_2_1;
vertex  vert2_2_2;
vertex  vert2_2_3;
vertex  vert2_2_4;
vertex  vert2_2_5;
vertex  vert2_2_6;

polygon  poly2_3;
vertex  vert2_3_1;
vertex  vert2_3_2;
vertex  vert2_3_3;
vertex  vert2_3_4;

polygon  poly2_4;
vertex  vert2_4_1;
vertex  vert2_4_2;
vertex  vert2_4_3;
vertex  vert2_4_4;

polygon  poly2_5;
vertex  vert2_5_1;
vertex  vert2_5_2;
vertex  vert2_5_3;
vertex  vert2_5_4;

polygon  poly2_6;
vertex  vert2_6_1;
vertex  vert2_6_2;
```

```
vertex  vert2_6_3;
vertex  vert2_6_4;
vertex  vert2_6_5;
vertex  vert2_6_6;
vertex  vert2_6_7;

polygon  poly2_7;
vertex  vert2_7_1;
vertex  vert2_7_2;
vertex  vert2_7_3;
vertex  vert2_7_4;
vertex  vert2_7_5;
vertex  vert2_7_6;
vertex  vert2_7_7;
vertex  vert2_7_8;
vertex  vert2_7_9;
vertex  vert2_7_10;
vertex  vert2_7_11;

polygon  poly2_8;
vertex  vert2_8_1;
vertex  vert2_8_2;
vertex  vert2_8_3;
vertex  vert2_8_4;

polygon  poly2_9;
vertex  vert2_9_1;
vertex  vert2_9_2;
vertex  vert2_9_3;
vertex  vert2_9_4;

polygon  poly2_10;
vertex  vert2_10_1;
vertex  vert2_10_2;
vertex  vert2_10_3;
vertex  vert2_10_4;
vertex  vert2_10_5;
vertex  vert2_10_6;
vertex  vert2_10_7;

polygon  poly2_11;
vertex  vert2_11_1;
vertex  vert2_11_2;
vertex  vert2_11_3;
vertex  vert2_11_4;

polygon  poly2_12;
vertex  vert2_12_1;
vertex  vert2_12_2;
vertex  vert2_12_3;
vertex  vert2_12_4;
```

```
polygon  poly2_13;
vertex   vert2_13_1;
vertex   vert2_13_2;
vertex   vert2_13_3;
vertex   vert2_13_4;
vertex   vert2_13_5;

polygon  poly2_14;
vertex   vert2_14_1;
vertex   vert2_14_2;
vertex   vert2_14_3;
vertex   vert2_14_4;
vertex   vert2_14_5;
vertex   vert2_14_6;

polygon  poly2_15;
vertex   vert2_15_1;
vertex   vert2_15_2;
vertex   vert2_15_3;
vertex   vert2_15_4;
vertex   vert2_15_5;

polygon  poly2_16;
vertex   vert2_16_1;
vertex   vert2_16_2;
vertex   vert2_16_3;
vertex   vert2_16_4;
vertex   vert2_16_5;

polygon  poly2_17;
vertex   vert2_17_1;
vertex   vert2_17_2;
vertex   vert2_17_3;
vertex   vert2_17_4;
vertex   vert2_17_5;
vertex   vert2_17_6;


/*declaring first node for graph*/

node* worldGraph_180_00;

/*delcaring structures for nodes and arcs*/

node  node2_1;
arc   arc2_1_1;

node  node2_2;
arc   arc2_2_1;
```

```
node  node2_3;
arc   arc2_3_1;
arc   arc2_3_2;

node  node2_4;
arc   arc2_4_1;

node  node2_5;
arc   arc2_5_1;
arc   arc2_5_2;
arc   arc2_5_3;

node  node2_6;
arc   arc2_6_1;
arc   arc2_6_2;

node  node2_7;
arc   arc2_7_1;
arc   arc2_7_2;
arc   arc2_7_3;
arc   arc2_7_4;
arc   arc2_7_5;

node  node2_8;
arc   arc2_8_1;
arc   arc2_8_2;
arc   arc2_8_3;

node  node2_9;
arc   arc2_9_1;
arc   arc2_9_2;

node  node2_10;
arc   arc2_10_1;
arc   arc2_10_2;

node  node2_11;
arc   arc2_11_1;
arc   arc2_11_2;

node  node2_12;
arc   arc2_12_1;
arc   arc2_12_2;

node  node2_13;
arc   arc2_13_1;
arc   arc2_13_2;

node  node2_14;
arc   arc2_14_1;
```

```
node  node2_15;
arc   arc2_15_1;

node  node2_16;
arc   arc2_16_1;

node  node2_17;
arc   arc2_17_1;

/*assigning values for polygons and vertices*/


InitializeWorld_180_00(){

 robotsWorld_180_00.polygonList = &poly2_1;

   strcpy(poly2_1.name, "C17");
   poly2_1.mode = -1;
   poly2_1.vertexList = &vert2_1_1;
   poly2_1.next = &poly2_2;
     vert2_1_1.posit.X = 5.60000;
     vert2_1_1.posit.Y = 3.80000;
     strcpy(vert2_1_1.bndry,"");
     vert2_1_1.next = &vert2_1_2;
     vert2_1_2.posit.X = 5.60000;
     vert2_1_2.posit.Y = 6.00000;
     strcpy(vert2_1_2.bndry,"h1");
     vert2_1_2.next = &vert2_1_3;
     vert2_1_2.prev = &vert2_1_1;
     vert2_1_3.posit.X = 8.40000;
     vert2_1_3.posit.Y = 6.00000;
     strcpy(vert2_1_3.bndry,"h1");
     vert2_1_3.next = &vert2_1_4;
     vert2_1_3.prev = &vert2_1_2;
     vert2_1_4.posit.X = 8.40000;
     vert2_1_4.posit.Y = 3.80000;
     strcpy(vert2_1_4.bndry,"h1");
     vert2_1_4.next = &vert2_1_5;
     vert2_1_4.prev = &vert2_1_3;
     vert2_1_5.posit.X = 7.40000;
     vert2_1_5.posit.Y = 3.80000;
     strcpy(vert2_1_5.bndry,"C15");
     vert2_1_5.next = &vert2_1_6;
     vert2_1_5.prev = &vert2_1_4;
     vert2_1_6.posit.X = 6.40000;
     vert2_1_6.posit.Y = 3.80000;
     strcpy(vert2_1_6.bndry,"h1");
     vert2_1_6.next = &vert2_1_1;
     vert2_1_6.prev = &vert2_1_5;
     vert2_1_1.prev = &vert2_1_6;
```

```
strcpy(poly2_2.name, "C16");
poly2_2.mode = -1;
poly2_2.vertexList = &vert2_2_1;
poly2_2.next = &poly2_3;
  vert2_2_1.posit.X = 0.00000;
  vert2_2_1.posit.Y = 3.80000;
  strcpy(vert2_2_1.bndry,"");
  vert2_2_1.next = &vert2_2_2;
  vert2_2_2.posit.X = 0.00000;
  vert2_2_2.posit.Y = 6.00000;
  strcpy(vert2_2_2.bndry,"h1");
  vert2_2_2.next = &vert2_2_3;
  vert2_2_2.prev = &vert2_2_1;
  vert2_2_3.posit.X = 4.20000;
  vert2_2_3.posit.Y = 6.00000;
  strcpy(vert2_2_3.bndry,"");
  vert2_2_3.next = &vert2_2_4;
  vert2_2_3.prev = &vert2_2_2;
  vert2_2_4.posit.X = 4.20000;
  vert2_2_4.posit.Y = 3.80000;
  strcpy(vert2_2_4.bndry,"h1");
  vert2_2_4.next = &vert2_2_5;
  vert2_2_4.prev = &vert2_2_3;
  vert2_2_5.posit.X = 2.00000;
  vert2_2_5.posit.Y = 3.80000;
  strcpy(vert2_2_5.bndry,"C12");
  vert2_2_5.next = &vert2_2_6;
  vert2_2_5.prev = &vert2_2_4;
  vert2_2_6.posit.X = 1.00000;
  vert2_2_6.posit.Y = 3.80000;
  strcpy(vert2_2_6.bndry,"h1");
  vert2_2_6.next = &vert2_2_1;
  vert2_2_6.prev = &vert2_2_5;
  vert2_2_1.prev = &vert2_2_6;

strcpy(poly2_3.name, "C14");
poly2_3.mode = -1;
poly2_3.vertexList = &vert2_3_1;
poly2_3.next = &poly2_4;
  vert2_3_1.posit.X = 4.40000;
  vert2_3_1.posit.Y = 3.60000;
  strcpy(vert2_3_1.bndry,"");
  vert2_3_1.next = &vert2_3_2;
  vert2_3_2.posit.X = 4.40000;
  vert2_3_2.posit.Y = 4.20000;
  strcpy(vert2_3_2.bndry,"h1");
  vert2_3_2.next = &vert2_3_3;
  vert2_3_2.prev = &vert2_3_1;
  vert2_3_3.posit.X = 5.40000;
  vert2_3_3.posit.Y = 4.20000;
  strcpy(vert2_3_3.bndry,"");
```

```
  vert2_3_3.next = &vert2_3_4;
  vert2_3_3.prev = &vert2_3_2;
  vert2_3_4.posit.X = 5.40000;
  vert2_3_4.posit.Y = 3.60000;
  strcpy(vert2_3_4.bndry,"C13");
  vert2_3_4.next = &vert2_3_1;
  vert2_3_4.prev = &vert2_3_3;
  vert2_3_1.prev = &vert2_3_4;

strcpy(poly2_4.name, "C15");
poly2_4.mode = -1;
poly2_4.vertexList = &vert2_4_1;
poly2_4.next = &poly2_5;
  vert2_4_1.posit.X = 6.40000;
  vert2_4_1.posit.Y = 3.60000;
  strcpy(vert2_4_1.bndry,"");
  vert2_4_1.next = &vert2_4_2;
  vert2_4_2.posit.X = 6.40000;
  vert2_4_2.posit.Y = 3.80000;
  strcpy(vert2_4_2.bndry,"C17");
  vert2_4_2.next = &vert2_4_3;
  vert2_4_2.prev = &vert2_4_1;
  vert2_4_3.posit.X = 7.40000;
  vert2_4_3.posit.Y = 3.80000;
  strcpy(vert2_4_3.bndry,"");
  vert2_4_3.next = &vert2_4_4;
  vert2_4_3.prev = &vert2_4_2;
  vert2_4_4.posit.X = 7.40000;
  vert2_4_4.posit.Y = 3.60000;
  strcpy(vert2_4_4.bndry,"C13");
  vert2_4_4.next = &vert2_4_1;
  vert2_4_4.prev = &vert2_4_3;
  vert2_4_1.prev = &vert2_4_4;

strcpy(poly2_5.name, "C12");
poly2_5.mode = -1;
poly2_5.vertexList = &vert2_5_1;
poly2_5.next = &poly2_6;
  vert2_5_1.posit.X = 1.00000;
  vert2_5_1.posit.Y = 3.60000;
  strcpy(vert2_5_1.bndry,"");
  vert2_5_1.next = &vert2_5_2;
  vert2_5_2.posit.X = 1.00000;
  vert2_5_2.posit.Y = 3.80000;
  strcpy(vert2_5_2.bndry,"C16");
  vert2_5_2.next = &vert2_5_3;
  vert2_5_2.prev = &vert2_5_1;
  vert2_5_3.posit.X = 2.00000;
  vert2_5_3.posit.Y = 3.80000;
  strcpy(vert2_5_3.bndry,"");
  vert2_5_3.next = &vert2_5_4;
```

```
  vert2_5_3.prev = &vert2_5_2;
  vert2_5_4.posit.X = 2.00000;
  vert2_5_4.posit.Y = 3.60000;
  strcpy(vert2_5_4.bndry,"C11");
  vert2_5_4.next = &vert2_5_1;
  vert2_5_4.prev = &vert2_5_3;
  vert2_5_1.prev = &vert2_5_4;

strcpy(poly2_6.name, "C13");
poly2_6.mode = -1;
poly2_6.vertexList = &vert2_6_1;
poly2_6.next = &poly2_7;
  vert2_6_1.posit.X = 2.00000;
  vert2_6_1.posit.Y = 3.60000;
  strcpy(vert2_6_1.bndry,"");
  vert2_6_1.next = &vert2_6_2;
  vert2_6_2.posit.X = 4.40000;
  vert2_6_2.posit.Y = 3.60000;
  strcpy(vert2_6_2.bndry,"C14");
  vert2_6_2.next = &vert2_6_3;
  vert2_6_2.prev = &vert2_6_1;
  vert2_6_3.posit.X = 5.40000;
  vert2_6_3.posit.Y = 3.60000;
  strcpy(vert2_6_3.bndry,"h1");
  vert2_6_3.next = &vert2_6_4;
  vert2_6_3.prev = &vert2_6_2;
  vert2_6_4.posit.X = 6.40000;
  vert2_6_4.posit.Y = 3.60000;
  strcpy(vert2_6_4.bndry,"C15");
  vert2_6_4.next = &vert2_6_5;
  vert2_6_4.prev = &vert2_6_3;
  vert2_6_5.posit.X = 7.40000;
  vert2_6_5.posit.Y = 3.60000;
  strcpy(vert2_6_5.bndry,"h1");
  vert2_6_5.next = &vert2_6_6;
  vert2_6_5.prev = &vert2_6_4;
  vert2_6_6.posit.X = 9.60000;
  vert2_6_6.posit.Y = 3.60000;
  strcpy(vert2_6_6.bndry,"");
  vert2_6_6.next = &vert2_6_7;
  vert2_6_6.prev = &vert2_6_5;
  vert2_6_7.posit.X = 9.60000;
  vert2_6_7.posit.Y = 3.60000;
  strcpy(vert2_6_7.bndry,"C11");
  vert2_6_7.next = &vert2_6_1;
  vert2_6_7.prev = &vert2_6_6;
  vert2_6_1.prev = &vert2_6_7;

strcpy(poly2_7.name, "C11");
poly2_7.mode = -1;
poly2_7.vertexList = &vert2_7_1;
```

```
poly2_7.next = &poly2_8;
  vert2_7_1.posit.X = 4.00000;
  vert2_7_1.posit.Y = 2.50000;
  strcpy(vert2_7_1.bndry,"C10");
  vert2_7_1.next = &vert2_7_2;
  vert2_7_2.posit.X = 0.00000;
  vert2_7_2.posit.Y = 2.50000;
  strcpy(vert2_7_2.bndry,"");
  vert2_7_2.next = &vert2_7_3;
  vert2_7_2.prev = &vert2_7_1;
  vert2_7_3.posit.X = 0.00000;
  vert2_7_3.posit.Y = 3.60000;
  strcpy(vert2_7_3.bndry,"h1");
  vert2_7_3.next = &vert2_7_4;
  vert2_7_3.prev = &vert2_7_2;
  vert2_7_4.posit.X = 1.00000;
  vert2_7_4.posit.Y = 3.60000;
  strcpy(vert2_7_4.bndry,"C12");
  vert2_7_4.next = &vert2_7_5;
  vert2_7_4.prev = &vert2_7_3;
  vert2_7_5.posit.X = 2.00000;
  vert2_7_5.posit.Y = 3.60000;
  strcpy(vert2_7_5.bndry,"C13");
  vert2_7_5.next = &vert2_7_6;
  vert2_7_5.prev = &vert2_7_4;
  vert2_7_6.posit.X = 9.60000;
  vert2_7_6.posit.Y = 3.60000;
  strcpy(vert2_7_6.bndry,"");
  vert2_7_6.next = &vert2_7_7;
  vert2_7_6.prev = &vert2_7_5;
  vert2_7_7.posit.X = 9.60000;
  vert2_7_7.posit.Y = 2.50000;
  strcpy(vert2_7_7.bndry,"h1");
  vert2_7_7.next = &vert2_7_8;
  vert2_7_7.prev = &vert2_7_6;
  vert2_7_8.posit.X = 8.60000;
  vert2_7_8.posit.Y = 2.50000;
  strcpy(vert2_7_8.bndry,"C9");
  vert2_7_8.next = &vert2_7_9;
  vert2_7_8.prev = &vert2_7_7;
  vert2_7_9.posit.X = 8.00000;
  vert2_7_9.posit.Y = 2.50000;
  strcpy(vert2_7_9.bndry,"h1");
  vert2_7_9.next = &vert2_7_10;
  vert2_7_9.prev = &vert2_7_8;
  vert2_7_10.posit.X = 7.40000;
  vert2_7_10.posit.Y = 2.50000;
  strcpy(vert2_7_10.bndry,"C8");
  vert2_7_10.next = &vert2_7_11;
  vert2_7_10.prev = &vert2_7_9;
  vert2_7_11.posit.X = 6.80000;
```

```
  vert2_7_11.posit.Y = 2.50000;
  strcpy(vert2_7_11.bndry,"h1");
  vert2_7_11.next = &vert2_7_1;
  vert2_7_11.prev = &vert2_7_10;
  vert2_7_1.prev = &vert2_7_11;

strcpy(poly2_8.name, "C9");
poly2_8.mode = -1;
poly2_8.vertexList = &vert2_8_1;
poly2_8.next = &poly2_9;
  vert2_8_1.posit.X = 8.60000;
  vert2_8_1.posit.Y = 2.30000;
  strcpy(vert2_8_1.bndry,"C4");
  vert2_8_1.next = &vert2_8_2;
  vert2_8_2.posit.X = 8.00000;
  vert2_8_2.posit.Y = 2.30000;
  strcpy(vert2_8_2.bndry,"");
  vert2_8_2.next = &vert2_8_3;
  vert2_8_2.prev = &vert2_8_1;
  vert2_8_3.posit.X = 8.00000;
  vert2_8_3.posit.Y = 2.50000;
  strcpy(vert2_8_3.bndry,"C11");
  vert2_8_3.next = &vert2_8_4;
  vert2_8_3.prev = &vert2_8_2;
  vert2_8_4.posit.X = 8.60000;
  vert2_8_4.posit.Y = 2.50000;
  strcpy(vert2_8_4.bndry,"");
  vert2_8_4.next = &vert2_8_1;
  vert2_8_4.prev = &vert2_8_3;
  vert2_8_1.prev = &vert2_8_4;

strcpy(poly2_9.name, "C8");
poly2_9.mode = -1;
poly2_9.vertexList = &vert2_9_1;
poly2_9.next = &poly2_10;
  vert2_9_1.posit.X = 6.80000;
  vert2_9_1.posit.Y = 2.30000;
  strcpy(vert2_9_1.bndry,"");
  vert2_9_1.next = &vert2_9_2;
  vert2_9_2.posit.X = 6.80000;
  vert2_9_2.posit.Y = 2.50000;
  strcpy(vert2_9_2.bndry,"C11");
  vert2_9_2.next = &vert2_9_3;
  vert2_9_2.prev = &vert2_9_1;
  vert2_9_3.posit.X = 7.40000;
  vert2_9_3.posit.Y = 2.50000;
  strcpy(vert2_9_3.bndry,"");
  vert2_9_3.next = &vert2_9_4;
  vert2_9_3.prev = &vert2_9_2;
  vert2_9_4.posit.X = 7.40000;
  vert2_9_4.posit.Y = 2.30000;
```

```
  strcpy(vert2_9_4.bndry,"C3");
  vert2_9_4.next = &vert2_9_1;
  vert2_9_4.prev = &vert2_9_3;
  vert2_9_1.prev = &vert2_9_4;

strcpy(poly2_10.name, "C10");
poly2_10.mode = -1;
poly2_10.vertexList = &vert2_10_1;
poly2_10.next = &poly2_11;
  vert2_10_1.posit.X = 0.00000;
  vert2_10_1.posit.Y = 2.50000;
  strcpy(vert2_10_1.bndry,"");
  vert2_10_1.next = &vert2_10_2;
  vert2_10_2.posit.X = 0.00000;
  vert2_10_2.posit.Y = 2.50000;
  strcpy(vert2_10_2.bndry,"C11");
  vert2_10_2.next = &vert2_10_3;
  vert2_10_2.prev = &vert2_10_1;
  vert2_10_3.posit.X = 4.00000;
  vert2_10_3.posit.Y = 2.50000;
  strcpy(vert2_10_3.bndry,"h1");
  vert2_10_3.next = &vert2_10_4;
  vert2_10_3.prev = &vert2_10_2;
  vert2_10_4.posit.X = 4.00000;
  vert2_10_4.posit.Y = 2.50000;
  strcpy(vert2_10_4.bndry,"C7");
  vert2_10_4.next = &vert2_10_5;
  vert2_10_4.prev = &vert2_10_3;
  vert2_10_5.posit.X = 3.20000;
  vert2_10_5.posit.Y = 2.50000;
  strcpy(vert2_10_5.bndry,"h1");
  vert2_10_5.next = &vert2_10_6;
  vert2_10_5.prev = &vert2_10_4;
  vert2_10_6.posit.X = 1.40000;
  vert2_10_6.posit.Y = 2.50000;
  strcpy(vert2_10_6.bndry,"C5");
  vert2_10_6.next = &vert2_10_7;
  vert2_10_6.prev = &vert2_10_5;
  vert2_10_7.posit.X = 0.80000;
  vert2_10_7.posit.Y = 2.50000;
  strcpy(vert2_10_7.bndry,"h1");
  vert2_10_7.next = &vert2_10_1;
  vert2_10_7.prev = &vert2_10_6;
  vert2_10_1.prev = &vert2_10_7;

strcpy(poly2_11.name, "C7");
poly2_11.mode = -1;
poly2_11.vertexList = &vert2_11_1;
poly2_11.next = &poly2_12;
  vert2_11_1.posit.X = 4.00000;
  vert2_11_1.posit.Y = 2.30000;
```

```
  strcpy(vert2_11_1.bndry,"C6");
  vert2_11_1.next = &vert2_11_2;
  vert2_11_2.posit.X = 3.20000;
  vert2_11_2.posit.Y = 2.30000;
  strcpy(vert2_11_2.bndry,"");
  vert2_11_2.next = &vert2_11_3;
  vert2_11_2.prev = &vert2_11_1;
  vert2_11_3.posit.X = 3.20000;
  vert2_11_3.posit.Y = 2.50000;
  strcpy(vert2_11_3.bndry,"C10");
  vert2_11_3.next = &vert2_11_4;
  vert2_11_3.prev = &vert2_11_2;
  vert2_11_4.posit.X = 4.00000;
  vert2_11_4.posit.Y = 2.50000;
  strcpy(vert2_11_4.bndry,"");
  vert2_11_4.next = &vert2_11_1;
  vert2_11_4.prev = &vert2_11_3;
  vert2_11_1.prev = &vert2_11_4;

strcpy(poly2_12.name, "C5");
poly2_12.mode = -1;
poly2_12.vertexList = &vert2_12_1;
poly2_12.next = &poly2_13;
  vert2_12_1.posit.X = 0.80000;
  vert2_12_1.posit.Y = 2.30000;
  strcpy(vert2_12_1.bndry,"");
  vert2_12_1.next = &vert2_12_2;
  vert2_12_2.posit.X = 0.80000;
  vert2_12_2.posit.Y = 2.50000;
  strcpy(vert2_12_2.bndry,"C10");
  vert2_12_2.next = &vert2_12_3;
  vert2_12_2.prev = &vert2_12_1;
  vert2_12_3.posit.X = 1.40000;
  vert2_12_3.posit.Y = 2.50000;
  strcpy(vert2_12_3.bndry,"");
  vert2_12_3.next = &vert2_12_4;
  vert2_12_3.prev = &vert2_12_2;
  vert2_12_4.posit.X = 1.40000;
  vert2_12_4.posit.Y = 2.30000;
  strcpy(vert2_12_4.bndry,"C1");
  vert2_12_4.next = &vert2_12_1;
  vert2_12_4.prev = &vert2_12_3;
  vert2_12_1.prev = &vert2_12_4;

strcpy(poly2_13.name, "C4");
poly2_13.mode = -1;
poly2_13.vertexList = &vert2_13_1;
poly2_13.next = &poly2_14;
  vert2_13_1.posit.X = 8.00000;
  vert2_13_1.posit.Y = 0.00000;
  strcpy(vert2_13_1.bndry,"h1");
```

```
  vert2_13_1.next = &vert2_13_2;
  vert2_13_2.posit.X = 8.00000;
  vert2_13_2.posit.Y = 2.30000;
  strcpy(vert2_13_2.bndry,"C9");
  vert2_13_2.next = &vert2_13_3;
  vert2_13_2.prev = &vert2_13_1;
  vert2_13_3.posit.X = 8.60000;
  vert2_13_3.posit.Y = 2.30000;
  strcpy(vert2_13_3.bndry,"");
  vert2_13_3.next = &vert2_13_4;
  vert2_13_3.prev = &vert2_13_2;
  vert2_13_4.posit.X = 9.60000;
  vert2_13_4.posit.Y = 2.30000;
  strcpy(vert2_13_4.bndry,"h1");
  vert2_13_4.next = &vert2_13_5;
  vert2_13_4.prev = &vert2_13_3;
  vert2_13_5.posit.X = 9.60000;
  vert2_13_5.posit.Y = 0.00000;
  strcpy(vert2_13_5.bndry,"h1");
  vert2_13_5.next = &vert2_13_1;
  vert2_13_5.prev = &vert2_13_4;
  vert2_13_1.prev = &vert2_13_5;

strcpy(poly2_14.name, "C3");
poly2_14.mode = -1;
poly2_14.vertexList = &vert2_14_1;
poly2_14.next = &poly2_15;
  vert2_14_1.posit.X = 5.00000;
  vert2_14_1.posit.Y = 0.00000;
  strcpy(vert2_14_1.bndry,"h1");
  vert2_14_1.next = &vert2_14_2;
  vert2_14_2.posit.X = 5.00000;
  vert2_14_2.posit.Y = 2.30000;
  strcpy(vert2_14_2.bndry,"h1");
  vert2_14_2.next = &vert2_14_3;
  vert2_14_2.prev = &vert2_14_1;
  vert2_14_3.posit.X = 6.80000;
  vert2_14_3.posit.Y = 2.30000;
  strcpy(vert2_14_3.bndry,"C8");
  vert2_14_3.next = &vert2_14_4;
  vert2_14_3.prev = &vert2_14_2;
  vert2_14_4.posit.X = 7.40000;
  vert2_14_4.posit.Y = 2.30000;
  strcpy(vert2_14_4.bndry,"h1");
  vert2_14_4.next = &vert2_14_5;
  vert2_14_4.prev = &vert2_14_3;
  vert2_14_5.posit.X = 7.80000;
  vert2_14_5.posit.Y = 2.30000;
  strcpy(vert2_14_5.bndry,"");
  vert2_14_5.next = &vert2_14_6;
  vert2_14_5.prev = &vert2_14_4;
```

```
  vert2_14_6.posit.X = 7.80000;
  vert2_14_6.posit.Y = 0.00000;
  strcpy(vert2_14_6.bndry,"h1");
  vert2_14_6.next = &vert2_14_1;
  vert2_14_6.prev = &vert2_14_5;
  vert2_14_1.prev = &vert2_14_6;

strcpy(poly2_15.name, "C6");
poly2_15.mode = -1;
poly2_15.vertexList = &vert2_15_1;
poly2_15.next = &poly2_16;
  vert2_15_1.posit.X = 3.20000;
  vert2_15_1.posit.Y = 2.30000;
  strcpy(vert2_15_1.bndry,"");
  vert2_15_1.next = &vert2_15_2;
  vert2_15_2.posit.X = 3.20000;
  vert2_15_2.posit.Y = 2.30000;
  strcpy(vert2_15_2.bndry,"C7");
  vert2_15_2.next = &vert2_15_3;
  vert2_15_2.prev = &vert2_15_1;
  vert2_15_3.posit.X = 4.00000;
  vert2_15_3.posit.Y = 2.30000;
  strcpy(vert2_15_3.bndry,"");
  vert2_15_3.next = &vert2_15_4;
  vert2_15_3.prev = &vert2_15_2;
  vert2_15_4.posit.X = 4.80000;
  vert2_15_4.posit.Y = 2.30000;
  strcpy(vert2_15_4.bndry,"h1");
  vert2_15_4.next = &vert2_15_5;
  vert2_15_4.prev = &vert2_15_3;
  vert2_15_5.posit.X = 4.80000;
  vert2_15_5.posit.Y = 2.30000;
  strcpy(vert2_15_5.bndry,"C2");
  vert2_15_5.next = &vert2_15_1;
  vert2_15_5.prev = &vert2_15_4;
  vert2_15_1.prev = &vert2_15_5;

strcpy(poly2_16.name, "C2");
poly2_16.mode = -1;
poly2_16.vertexList = &vert2_16_1;
poly2_16.next = &poly2_17;
  vert2_16_1.posit.X = 2.40000;
  vert2_16_1.posit.Y = 0.00000;
  strcpy(vert2_16_1.bndry,"h1");
  vert2_16_1.next = &vert2_16_2;
  vert2_16_2.posit.X = 2.40000;
  vert2_16_2.posit.Y = 2.30000;
  strcpy(vert2_16_2.bndry,"h1");
  vert2_16_2.next = &vert2_16_3;
  vert2_16_2.prev = &vert2_16_1;
  vert2_16_3.posit.X = 3.20000;
```

```
  vert2_16_3.posit.Y = 2.30000;
  strcpy(vert2_16_3.bndry,"C6");
  vert2_16_3.next = &vert2_16_4;
  vert2_16_3.prev = &vert2_16_2;
  vert2_16_4.posit.X = 4.80000;
  vert2_16_4.posit.Y = 2.30000;
  strcpy(vert2_16_4.bndry,"");
  vert2_16_4.next = &vert2_16_5;
  vert2_16_4.prev = &vert2_16_3;
  vert2_16_5.posit.X = 4.80000;
  vert2_16_5.posit.Y = 0.00000;
  strcpy(vert2_16_5.bndry,"h1");
  vert2_16_5.next = &vert2_16_1;
  vert2_16_5.prev = &vert2_16_4;
  vert2_16_1.prev = &vert2_16_5;

strcpy(poly2_17.name, "C1");
poly2_17.mode = -1;
poly2_17.vertexList = &vert2_17_1;
poly2_17.next = &poly2_1;
  vert2_17_1.posit.X = 0.00000;
  vert2_17_1.posit.Y = 0.00000;
  strcpy(vert2_17_1.bndry,"h1");
  vert2_17_1.next = &vert2_17_2;
  vert2_17_2.posit.X = 0.00000;
  vert2_17_2.posit.Y = 2.30000;
  strcpy(vert2_17_2.bndry,"h1");
  vert2_17_2.next = &vert2_17_3;
  vert2_17_2.prev = &vert2_17_1;
  vert2_17_3.posit.X = 0.80000;
  vert2_17_3.posit.Y = 2.30000;
  strcpy(vert2_17_3.bndry,"C5");
  vert2_17_3.next = &vert2_17_4;
  vert2_17_3.prev = &vert2_17_2;
  vert2_17_4.posit.X = 1.40000;
  vert2_17_4.posit.Y = 2.30000;
  strcpy(vert2_17_4.bndry,"h1");
  vert2_17_4.next = &vert2_17_5;
  vert2_17_4.prev = &vert2_17_3;
  vert2_17_5.posit.X = 2.20000;
  vert2_17_5.posit.Y = 2.30000;
  strcpy(vert2_17_5.bndry,"");
  vert2_17_5.next = &vert2_17_6;
  vert2_17_5.prev = &vert2_17_4;
  vert2_17_6.posit.X = 2.20000;
  vert2_17_6.posit.Y = 0.00000;
  strcpy(vert2_17_6.bndry,"h1");
  vert2_17_6.next = &vert2_17_1;
  vert2_17_6.prev = &vert2_17_5;
  vert2_17_1.prev = &vert2_17_6;
}
```

```
/*assigning values for node and arcs*/


InitializeGraph_180_00(){

 worldGraph_180_00 = &node2_1;
  node2_1.cell = &poly2_1;
  node2_1.arcList = &arc2_1_1;
  node2_1.predecessor = NULL;
  node2_1.curArc = NULL;
  node2_1.next = &node2_2;
    arc2_1_1.Node = &node2_3;
    arc2_1_1.visited = 0;
    arc2_1_1.next = NULL;
  node2_2.cell = &poly2_2;
  node2_2.arcList = &arc2_2_1;
  node2_2.predecessor = NULL;
  node2_2.curArc = NULL;
  node2_2.next = &node2_3;
    arc2_2_1.Node = &node2_6;
    arc2_2_1.visited = 0;
    arc2_2_1.next = NULL;
  node2_3.cell = &poly2_4;
  node2_3.arcList = &arc2_3_1;
  node2_3.predecessor = NULL;
  node2_3.curArc = NULL;
  node2_3.next = &node2_4;
    arc2_3_1.Node = &node2_1;
    arc2_3_1.visited = 0;
    arc2_3_1.next = &arc2_3_2;
    arc2_3_2.Node = &node2_5;
    arc2_3_2.visited = 0;
    arc2_3_2.next = NULL;
  node2_4.cell = &poly2_3;
  node2_4.arcList = &arc2_4_1;
  node2_4.predecessor = NULL;
  node2_4.curArc = NULL;
  node2_4.next = &node2_5;
    arc2_4_1.Node = &node2_5;
    arc2_4_1.visited = 0;
    arc2_4_1.next = NULL;
  node2_5.cell = &poly2_6;
  node2_5.arcList = &arc2_5_1;
  node2_5.predecessor = NULL;
  node2_5.curArc = NULL;
  node2_5.next = &node2_6;
    arc2_5_1.Node = &node2_3;
    arc2_5_1.visited = 0;
    arc2_5_1.next = &arc2_5_2;
    arc2_5_2.Node = &node2_4;
```

```
    arc2_5_2.visited = 0;
    arc2_5_2.next = &arc2_5_3;
    arc2_5_3.Node = &node2_7;
    arc2_5_3.visited = 0;
    arc2_5_3.next = NULL;
  node2_6.cell = &poly2_5;
  node2_6.arcList = &arc2_6_1;
  node2_6.predecessor = NULL;
  node2_6.curArc = NULL;
  node2_6.next = &node2_7;
    arc2_6_1.Node = &node2_2;
    arc2_6_1.visited = 0;
    arc2_6_1.next = &arc2_6_2;
    arc2_6_2.Node = &node2_7;
    arc2_6_2.visited = 0;
    arc2_6_2.next = NULL;
  node2_7.cell = &poly2_7;
  node2_7.arcList = &arc2_7_1;
  node2_7.predecessor = NULL;
  node2_7.curArc = NULL;
  node2_7.next = &node2_8;
    arc2_7_1.Node = &node2_5;
    arc2_7_1.visited = 0;
    arc2_7_1.next = &arc2_7_2;
    arc2_7_2.Node = &node2_6;
    arc2_7_2.visited = 0;
    arc2_7_2.next = &arc2_7_3;
    arc2_7_3.Node = &node2_9;
    arc2_7_3.visited = 0;
    arc2_7_3.next = &arc2_7_4;
    arc2_7_4.Node = &node2_10;
    arc2_7_4.visited = 0;
    arc2_7_4.next = &arc2_7_5;
    arc2_7_5.Node = &node2_8;
    arc2_7_5.visited = 0;
    arc2_7_5.next = NULL;
  node2_8.cell = &poly2_10;
  node2_8.arcList = &arc2_8_1;
  node2_8.predecessor = NULL;
  node2_8.curArc = NULL;
  node2_8.next = &node2_9;
    arc2_8_1.Node = &node2_7;
    arc2_8_1.visited = 0;
    arc2_8_1.next = &arc2_8_2;
    arc2_8_2.Node = &node2_11;
    arc2_8_2.visited = 0;
    arc2_8_2.next = &arc2_8_3;
    arc2_8_3.Node = &node2_13;
    arc2_8_3.visited = 0;
    arc2_8_3.next = NULL;
  node2_9.cell = &poly2_8;
```

```
node2_9.arcList = &arc2_9_1;
node2_9.predecessor = NULL;
node2_9.curArc = NULL;
node2_9.next = &node2_10;
  arc2_9_1.Node = &node2_7;
  arc2_9_1.visited = 0;
  arc2_9_1.next = &arc2_9_2;
  arc2_9_2.Node = &node2_14;
  arc2_9_2.visited = 0;
  arc2_9_2.next = NULL;
node2_10.cell = &poly2_9;
node2_10.arcList = &arc2_10_1;
node2_10.predecessor = NULL;
node2_10.curArc = NULL;
node2_10.next = &node2_11;
  arc2_10_1.Node = &node2_7;
  arc2_10_1.visited = 0;
  arc2_10_1.next = &arc2_10_2;
  arc2_10_2.Node = &node2_15;
  arc2_10_2.visited = 0;
  arc2_10_2.next = NULL;
node2_11.cell = &poly2_11;
node2_11.arcList = &arc2_11_1;
node2_11.predecessor = NULL;
node2_11.curArc = NULL;
node2_11.next = &node2_12;
  arc2_11_1.Node = &node2_8;
  arc2_11_1.visited = 0;
  arc2_11_1.next = &arc2_11_2;
  arc2_11_2.Node = &node2_12;
  arc2_11_2.visited = 0;
  arc2_11_2.next = NULL;
node2_12.cell = &poly2_15;
node2_12.arcList = &arc2_12_1;
node2_12.predecessor = NULL;
node2_12.curArc = NULL;
node2_12.next = &node2_13;
  arc2_12_1.Node = &node2_11;
  arc2_12_1.visited = 0;
  arc2_12_1.next = &arc2_12_2;
  arc2_12_2.Node = &node2_16;
  arc2_12_2.visited = 0;
  arc2_12_2.next = NULL;
node2_13.cell = &poly2_12;
node2_13.arcList = &arc2_13_1;
node2_13.predecessor = NULL;
node2_13.curArc = NULL;
node2_13.next = &node2_14;
  arc2_13_1.Node = &node2_8;
  arc2_13_1.visited = 0;
  arc2_13_1.next = &arc2_13_2;
```

```
    arc2_13_2.Node = &node2_17;
    arc2_13_2.visited = 0;
    arc2_13_2.next = NULL;
  node2_14.cell = &poly2_13;
  node2_14.arcList = &arc2_14_1;
  node2_14.predecessor = NULL;
  node2_14.curArc = NULL;
  node2_14.next = &node2_15;
    arc2_14_1.Node = &node2_9;
    arc2_14_1.visited = 0;
    arc2_14_1.next = NULL;
  node2_15.cell = &poly2_14;
  node2_15.arcList = &arc2_15_1;
  node2_15.predecessor = NULL;
  node2_15.curArc = NULL;
  node2_15.next = &node2_16;
    arc2_15_1.Node = &node2_10;
    arc2_15_1.visited = 0;
    arc2_15_1.next = NULL;
  node2_16.cell = &poly2_16;
  node2_16.arcList = &arc2_16_1;
  node2_16.predecessor = NULL;
  node2_16.curArc = NULL;
  node2_16.next = &node2_17;
    arc2_16_1.Node = &node2_12;
    arc2_16_1.visited = 0;
    arc2_16_1.next = NULL;
  node2_17.cell = &poly2_17;
  node2_17.arcList = &arc2_17_1;
  node2_17.predecessor = NULL;
  node2_17.curArc = NULL;
   node2_17.next = NULL;
    arc2_17_1.Node = &node2_13;
    arc2_17_1.visited = 0;
    arc2_17_1.next = NULL;
}
```

G. HOMOTOPY CLASSES

1. cells.h

```
/*****************************************************************
FILE: cells.h
PURPOSE: This file contains the prototypes for the functions
      which are used to  determine the homotopy classes

*****************************************************************/

#ifndef __cells_h
#define __cells_h
```

```
#include "build.h"

/*********************insideCell(polygon* point)************************
This function determines whether the input point is inside of the
convex polygon.  It can be used to verify a suspected location, or
called for every cell until a match is found.

*************************************************************************/
int insideCell(polygon*, point);

/*****************findCell(world*, point*)*****************************
This function calls insideCell() for every cell in the world until
true is returned.  It then returns a pointer to that cell

*************************************************************************/
polygon* findCell(world*, point);

/*******findHmtpClss(node*, world*, point, point)*************************
This function finds all the homotopy class from one point to the
other by searching the connectity graph.

*************************************************************************/
void findHmtpClss(node*, world*, point, point);

/************dpthFstSch(node*, node*, node*)*****************************
This function performs  a depth first search, and prints out the
path found.  The function is called recursively.

*************************************************************************/
void dpthFstSch(node*, polygon*, polygon*);

#endif
```

2. cells.c

```
/*************************************************************
FILE: cells.c
PURPOSE: This file contains the functions which are used to
      determine the homotopy classes

*************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "build.h"
#include "util.h"

static numberOfClasses;

/*********************insideCell(polygon* point)************************
This function determines whether the input point is inside of the
```

```
convex polygon.  It can be used to verify a suspected location, or
called for every cell until a match is found.

**********************************************************************/
int
insideCell(polygon* cnvxCell, point robotLocation){

   vertex* curVertex;

   curVertex = cnvxCell->vertexList;

   do{/*check all the vertices*/
      if(order(robotLocation,curVertex->posit,curVertex->next->posit) < 0.0)
         curVertex = curVertex->next;
      else
         return 0;/*made a left turn*/
   }while(curVertex != cnvxCell->vertexList);

   return 1;/*all right turns; inside cell*/
}
/***************findCell(world*, point*)*******************************
This function calls insideCell() for every cell in the world until
true is returned.  It then returns a pointer to that cell

**********************************************************************/
polygon*
findCell(world* decompWorld, point robotLocation){

   polygon* curPolygon;

   curPolygon = decompWorld->polygonList;

   do{/*check all cells*/
      if (insideCell(curPolygon, robotLocation))
         return curPolygon;
      else
         curPolygon = curPolygon->next;
   }while(curPolygon != decompWorld->polygonList);

   return NULL;/*point is outside of world*/
}
/***********dpthFstSch(node*, node*, node*)*****************************
This function performs  a depth first search, and prints out the
path found.  The function is called recursively.

**********************************************************************/
void
dpthFstSch(node* cGraph, node* robNode, node* goalNode){

   node* curNode;
   node* printNode = NULL;
```

```
    extern int numberOfClasses;

    if(robNode == goalNode){/* we are at the goal */
      printf("The cell movement sequence for homotopy class number %d is\n",
                    ++numberOfClasses);
      /* this block of code just prints out the path from the
        start node to the goal node */
      curNode = goalNode;
      while(curNode->predecessor != curNode)
        curNode = curNode->predecessor;
      printf("%s",curNode->cell->name);
      while(curNode != goalNode){
        printNode = goalNode;
        while(printNode->predecessor != curNode)
          printNode = printNode->predecessor;
        printf("->%s",printNode->cell->name);
        curNode = printNode;
}
      printf("\n");


    }else{
      while (robNode->curArc){/* for all nodes adj to this one */
        if(!(robNode->curArc->Node->predecessor)){/*has not been checked */

          robNode->curArc->Node->predecessor = robNode;

          dpthFstSch(cGraph, robNode->curArc->Node, goalNode);

          /* these are reset to allow for backtracking */
          robNode->curArc->Node->curArc = robNode->curArc->Node->arcList;
          robNode->curArc->Node->predecessor = NULL;
        }
        robNode->curArc = robNode->curArc->next;
      }
    }
}

/******findHmtpClss(node*, world*, point, point)*****************************
This function finds all the homotopy class from one point to the
other by searching the connectity graph.

*****************************************************************************/
void
findHmtpClss(node* cGraph, world* robWorld, point robLoc, point goalLoc){

    polygon* robCell;
    polygon* goalCell;

    node* curNode;
    node* robNode;
```

```
    node* goalNode;

    robCell = findCell(robWorld, robLoc);
    goalCell = findCell(robWorld, goalLoc);
    numberOfClasses = 0;

    /* initialize graph */
    curNode = cGraph;
    while(curNode){
      curNode->predecessor = NULL;
      curNode->curArc = curNode->arcList;
      curNode = curNode->next;
    }

    if(robCell == goalCell)
      printf("robot and goal are in the same cell");
    else{
      /*find appropriate node on graph for robot's location*/
      robNode = cGraph;
      while(robNode->cell != robCell)
        robNode = robNode->next;
      robNode->predecessor = robNode;

      /*find appropriate node on graph for goal's location*/
      goalNode = cGraph;
      while(goalNode->cell != goalCell)
        goalNode = goalNode->next;

      dpthFstSch(cGraph, robNode, goalNode);
    }
}
```

APPENDIX B

This appendix is a users guide for the decomposition programs implemented as part of this thesis. It assumes that the user has a working knowledge of Yamabico's kernel.

A. CREATING VERTEX FILE

The vertex file must be of the following form:

world_name(polygon_name_1(mode(x_{11},y_{11})(x_{12},y_{12})...(x_{1n},y_{1n}))
(polygon_name_2(mode(x_{21},y_{21})(x_{22},y_{22})...(x_{2m},y_{2m}))
...
(polygon_name_k(mode(x_{k1},y_{k1})(x_{k2},y_{k2})...(x_{kl},y_{kl})))

where:

world_name is a string of 15 or less characters.

polygon_name_i is a string of 5 or less characters.

mode is 1 for normal holes, and -1 for inverted holes.

(x_{i1},y_{i2})(x_{i2},y_{i2})...(x_{in},y_{in}) is an ordered list of vertices

white space is ignored.

B. RUNNING DECOMPOSE PROGRAM

The command line for the DecomposeWorld program is:

DecomposeWorld <inputfile> [number of sweeps] [sweep angle list]

where:

<inputfile> is the name of the file containing the world vertices and is of the form described in the first paragraph. This entry is mandatory.

[number of sweeps] is the number of distinct decompositions desired for this world. This default value is 1.

[sweep angle list] is a list of sweep angles. The number of entries in this list must match the number of sweeps desired. Each entry must be in the range (0.0..180.0]. Entries must be separated by whitespace. The default value is 90.0.

If only one sweep is performed, or the default values are used, then the program creates a file named *plot.dat* which contains the cell vertices in a format that can be used by GnuPlot.

C. DECOMPOSED WORLD MODELS

Each decomposition performed by DecomposeWorld creates a C file that is tagged with the sweep angle. Each file contains the declarations for the robot's world and the connectivity graph associated with the decomposition. Additionally, each C file contains two functions which initialize these declarations when needed. In order to use these data structures, the following steps must be performed:

1. Compile and link each *decomposeworld_xxx_xx.c* file as a part of building the kernel.
2. Include an extern declaration for the robot's world and the connectivity graph for each representation. They should look like this:

```
extern world    robotsWorld_xxx_xx;
extern node*    worldGraph_xxx_xx;
```

3. Before using the world information, call the following functions for each representation:

```
InitializeWorld_xxx_xx();
InitializeGraph_xxx_xx();
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
4.	Chairman, CodeMA Department of Mathematics Naval Postgraduate School Monterey, CA 93943	1
5.	Professor Yutaka Kanayama, Code CS/Ka Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
6.	Professor Craig W. Rasmussen, Code MA/Ra Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5000	2
7.	CPT Thomas G. Hopkins 2816 Pulpit Hill Woodbridge, VA 22191	1
8.	CPT Timothy A. Haight 1710 I'on Avenue Sullivan's Isle, SC 29482	4